VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Komprimace XML dat

Compression of XML data

2018 Martin Krča

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Martin Krča**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Komprimace XML dat
Compression of XML data

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je nastudovat možnosti komprimace XML dat, vybrané algoritmy naimplementovat a porovnat jak z pohledu komprimačního poměru, tak z pohledu času komprimace i dekomprimace.
Práce bude obsahovat následující body:
1. Rešerše metod pro komprimaci XML dat.
2. Popis a implementace vybraných metod.
3. Experimentální srovnání z pohledu komprimačního poměru a času komprimace i dekomprimace.

Seznam doporučené odborné literatury:

[1] SALOMON, David. Data compression: the complete reference. Springer Science & Business Media, 2004.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Petr Lukáš**

Datum zadání: 01.09.2017
Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2018

.....

Velmi děkuji vedoucímu práce Ing. Petrovi Lukášovi za jeho ochotu, trpělivost, cenné rady a pomoc a vedení mne skrze jeden z nejdůležitějších kroků na mé akademické cestě.

Abstrakt

Tato bakalářská práce se zabývá problematikou komprese a dekomprese XML souborů a jednotlivých kompresních algoritmů a metod.

Klíčová slova: Bakalářská práce, C++, XML, Komprese, Dekomprese, Data

Abstract

This bachelor thesis is about problematics of compression and decompression of XML files and about particular compression algorithms and methods.

Key Words: Bachelor thesis, C++, XML, Compression, Decompression, Data

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

XML je v dnešní době velmi rozšířený formát pro výměnu dat. Největší výhodou tohoto datového formátu je jeho univerzálnost a přenositelnost mezi systémy. Jeho nevýhodou je však náročnost na fyzické úložiště. Tento problém se dá řešit právě pomocí komprese dat. Jelikož je XML v podstatě textový soubor, lze na něj aplikovat běžné kompresní algoritmy. Pro lepší výsledek komprese je však nutné použít metody specializující se právě na kompresi XML dokumentů. V této bakalářské práci se budeme zabývat jednotlivými základními principy komprese, speciálně pak metodami pro kompresi právě XML dokumentů. Cílem této bakalářské práce je jednotlivé kompresní metody popsat, naimplementovat a otestovat. Součástí práce je experimentální srovnání, a to jak z pohledu času potřebného ke kompresi a dekompresi, tak s ohledem na úsporu místa na fyzickém uložišti.

1.1 Struktura práce

První kapitola je věnována charakteristice datového formátu XML. Je zde vysvětleno, co to XML je, k čemu se používá, jaké jsou jeho výhody oproti jiným datovým formátům a naopak jsou zde popsány i jeho nedostatky. Dále se zmiňujeme o možnostech a způsobech zpracování XML dokumentů a vysvětlujeme různé principy přístupu k jejich zpracování.

Druhá kapitola je věnována kompresi dat obecně. Jsou zde popsány základní kompresní principy a algoritmy, které jsou používány pro jakékoliv druhy dat. Dále jsou zde popsány kompresní metody, které neberou ohled na strukturu XML souborů, ale i tak mohou dosahovat dobrých výsledků komprese. Jelikož je však tato právě věnována kompresi XML dokumentů, věnujeme se v této kapitole především kompresním nástrojům s ohledem na strukturu XML. Popisujeme jak nástroje, které podporují dotazování nad komprimovanými daty, tak nástroje, které je potřeba pro možnost dotazování dekomprimovat.

V třetí kapitole se věnujeme implementaci jednotlivých algoritmů do testovací aplikace. Popisujeme návrh aplikace a jednotlivé třídy, se kterými aplikace pracuje. Možnosti interakce s aplikací jsou poté dále rozvedeny v příloze A.

V poslední kapitole se věnujeme právě testování komprese a dekomprese XML dokumentů a jednotlivým výsledkům. Lze zde nalézt popis testovacího stroje, parametrů pro testování, jako jsou vybrané naimplementované kompresní metody a popis XML kolekcí, na kterých byla komprese testována. Dále se zde nachází výsledky a především porovnání jednotlivých algoritmů a to jak z pohledu času potřebného pro vykonání komprese, tak podle množství uspořeného místa.

2 XML

XML neboli eXtensible Markup Language je obecný značkovací jazyk, který byl vyvinut a standardizován konsorciem W3C. Tento jazyk vznikl v roce 1996, avšak jeho verze 1.0 byla standardizována až v roce 1998 [5]. V roce 2001 se objevila verze 1.1 [6], která přinesla možnost použití více znaků, jelikož verze 1.0 používala pouze znaky Unicode 2.0. Toto omezení však platí pouze na použití znaků v elementech a atributech, jinak lze v obou verzích použít jakékoliv znaky. V současné době se tak používá pátá edice verze 1.0 z roku 2008 a druhá edice verze 1.1 z roku 2006. XML se používá především pro serializaci a výměnu dat a jeho hlavní výhodou je nezávislost na programovacím jazyce používané aplikace.

XML je jednoduchý obecný jazyk, který umožňuje vytvářet vlastní značkovací jazyky. Byl založen na jazyku GML (Standart Generalized Markup Language) a je jeho podmnožinou. Lze si jej tedy představit jako sadu předpisů a pravidel pro vytváření konkrétních jazyků. Hlavní využití nachází jako uložiště dat, které je přenositelné mezi platformami. Jelikož si lze jednotlivé prvky pojmenovat jakýmkoliv způsobem, XML nachází další využití ve formě databází či konfiguračních souborů. Struktura XML dokumentu se skládá ze značek a hodnot. Kód je tak jednoduše čitelný i pro člověka bez jakéhokoliv zpracování. Díky této vlastnosti je však tento jazyk relativně neúsporný z pohledu úložného prostoru.

2.1 Charakteristika

XML se dá chápat jako textový dokument, jelikož je tvořen posloupností znaků sady Unicode. XML nedefinuje vzhled dokumentu, ale jeho obsah. Zároveň se dá XML dokument chápat jako kořenový strom s ohodnocením vrcholů. Každý XML soubor začíná označením verze XML a volitelně označením sady znaků pro kódování. Dále už v dokumentu nalezneme pouze data, která jsou rozdělena na značky a obsah. Značky jsou uvozeny znakem ”<” a končí znakem ”>”. Koncová značka pak začíná ”</” a končí ”>”. Cokoliv je mezi značkami, je bráno jako obsah. Syntaxe souboru musí splňovat pravidla „Správně utvořeného dokumentu XML“ [3]. Mezi tato pravidla patří například, že se značky mohou vnořovat, ale nesmí se křížit. Dokument musí mít právě jeden kořenový element a všechny hodnoty atributů musí být uzavřeny v uvozovkách. Ukázku XML dokumentu můžete vidět na Výpisu č. 1.

```
<?xml version="1.0" encoding="utf-8" ?>.
<Knihy>
   <Kniha>
      <KnihaID> 0001 </KnihaID>
      <Nazev> The Old Man and the Sea </Nazev>
      <Autor>
         <Jmeno> Ernest </Jmeno>
         <Prijmeni> Hemingway </Prijmeni>
```

```
      </Autor>
      <Cena jednotka="Kč"> 150 </Cena>
   </Kniha>
<!-- Poznámka: Přidat více knih. -->
</Knihy>
```

Výpis 1: Struktura dokumentu XML

2.2 Výhody

- Není třeba vytvářet žádné převodní rozhraní mezi systémy. Pouze se definuje struktura dokumentu pro jednoduchou přenositelnost mezi systémy.
- XML dokument popisuje jména datových položek, jejich strukturu a hodnoty, je tak lehce čitelný pro člověka.
- K úpravě XML dokumentů lze použít běžné textové editory, jelikož lze snadno zjistit vnitřní strukturu dokumentu.
- Vhodné při vývoji software, u kterého není předem známo, na které platformě a které formáty bude využívat.
- Díky využití znakové sady Unicode lze použít znaky ze všech světových jazyků.
- Díky možnosti vytvářet vlastní značky je XML vhodný například pro konfigurační soubory, databáze či seznam položek v internetovém obchodu, jelikož je jazyk rozšiřitelný na XHTML.

2.3 Nevýhody

- Díky tomu, že informuje přímo o významu, druhu a obsahu dat, je tento jazyk výřečný. Tímto tak kladou XML dokumenty zvýšené nároky na jejich fyzické uložiště.
- Další nevýhodou je pak nutnost syntaktické analýzy při zpracování dat. Při použití principu DOM (popsaný v kapitole 2.4.1) k serializaci dat se tak nejprve musí vytvořit struktura dokumentu a teprve až poté z něj lze získávat data. Tomuto problému se však dá předejít při přístupu k souboru pomocí Simple API for XML, který bude podrobněji popsán v kapitole 2.4.2.
- Jelikož má XML textovou podobu, je zde problém zachycení binárních dat. Tento problém se však dá vyřešit například pomocí kódování Base64 [15].

2.4 Zpracování

Pro práci s XML soubory je nutné je nejdříve analyzovat a poté z nich lze získávat data. Dvěma nejčastějšími přístupy ke zpracování souboru XML jsou metody DOM a SAX, viz kapitoly 2.4.1 a 2.4.2. Základem, aby bylo možné soubor správně analyzovat, je povinnost souboru být v souladu s pravidly „Správně utvořeného dokumentu XML“.

2.4.1 Document Object Model

DOM je platformně a jazykově neutrální rozhraní, které umožňuje programům a skriptům dynamicky přistupovat a aktualizovat obsah, strukturu a styl dokumentů. Jedná se o objektově orientovanou reprezentaci XML dokumentu. DOM přišel jako sjednocení předchozích rozhraní pro manipulaci s webovými dokumenty. Přistupuje k dokumentu jako ke stromu a tím využívá stromovou strukturu XML. Při čtení souboru a vytváření struktury tedy postupuje od kořene po jednotlivých uzlech až na listové uzly. Využití této metody vyžaduje nahrání celého souboru do paměti, což může zvýšit paměťovou i časovou náročnost systému. Po nahrání do systému je již však práce s daty rychlá. Tento postup je tedy vhodný tam, kde je předpoklad opakovaného či náhodného přístupu k datům, stejně tak pokud se předpokládá spíše menší velikost XML [10]. Uvažujme XML dokument na Výpisu číslo 1. Příklad využití tohoto přístupu nalezenete na Výpisu č. 2

```
Document doc = DocumentBuilder.parse(inputFile);
NodeList nList = doc.getElementsByTagName("kniha");
for (int i = 0; i < nList.getLength(); i++)
{
   Node nNode = nList.item(i);
   if (nNode.getNodeType() == Node.ELEMENT_NODE)
   {
      Element eElement = (Element) nNode;
      System.out.println("Jmeno autora : "
      + eElement.getElementsByTagName("jmeno").item(0).getTextContent());
      System.out.println("Měna prodávané knihy je : "
      + eElement.getAttribute("jednotka"));
   }
}
```

Výpis 2: Čtení hodnot pomocí přístupu DOM

2.4.2 Simple API for XML

SAX je aplikační rozhraní pro zpracovávání XML dokumentů pracující na bázi řešení událostí. Umožňuje programům a skriptům sekvenčně přistupovat k obsahu XML souborů. Tato metoda

čte soubor postupně, rozdělí jej na jednotlivé menší části jako jsou značky, atributy, obsahy elementů a podobně. Při nalezení jednotlivých částí se poté spouštějí události a už je pouze na programátorovi, jak s nimi naloží. Za tímto účelem je obvykle nutné implementovat nějakou abstraktní třídu tak, že se přetíží metody obsluhující jednotlivé události. Využití této metody však nevyžaduje nahrávání struktury celého souboru do paměti, díky čemuž je několikrát rychlejší než DOM. Toho se využívá především u větších XML dokumentů, u kterých není potřeba přistupovat k souboru náhodně, ale sekvenčně bez okamžité možnosti zápisu. Toto API bylo nejprve vytvořeno pro programovací jazyk Java. V dnešní době už však existuje více verzí, které podporují několik dalších programovacích jazyků [1]. Příklad využití tohoto přístupu nalezenete na Výpisu č. 3

```
void OnElementStart(string& element, SaxParserAttributes& attributes)
{
   cout << "Element=" << element << endl;        // <element>
}
void OnElementEnd(string& element)
{
   cout << "</" << element << ">" << std::endl;  //</element>
}
void OnCharacterData(string& characterData)
{
   cout << "Data=" << characterData << std::endl;  //<>data</>
}
virtual void OnComment(string& comment)
{
   cout << "comment=" << comment << std::endl;   // <!-- poznámka -->
}
```

Výpis 3: Čtení XML souboru pomocí přístupu SAX

3 Komprese dat

Kompresi lze obecně označit jako operaci, při které se zmenšuje objem dat, přičemž obsažená informace zůstává stejná. Cílem této operace je zmenšit množství potřebného prostoru pro uložení dat (paměť), či snížit nároky na přenosové médium (zkrátit dobu přenosu souborů). Jak již bylo zmíněno výše, XML není navržen s ohledem na úsporu místa. V XML se například objevují sémantické značky se stejnou informací, např. že se jedná o knihu pomocí <book></book>.

Kompresi dat lze klasifikovat do dvou základních typů [14] a to:

- **Ztrátová komprese** je typ komprese dat, při které se zmenšuje objem dat smazáním určitých nepotřebných informací. Po kompresi již nemůžeme dekompresí zrekonstruovat původní informaci. Tento typ komprese se využívá především tam, kde výhoda zmenšeného souboru je větší, než nevýhoda ztráty informace. Využívá se tedy například pro kompresi obrazových a zvukových souborů.
- **Bezeztrátová komprese** je typ komprese dat, při které se zmenšuje objem dat při zachování všech informací. Oproti ztrátové kompresi není úspora místa tak velká, avšak je zachována celá informace i po dekompresi. Tohoto se využívá především tam, kde by ztráta jakékoliv informace znamenala poškození souboru. Využívá se tedy především v informačních technologiích. V této práci se budeme zaměřovat pouze na tento typ komprese.

Míru komprese můžeme vyčíslit pomocí tzv. *kompresního poměru*, který je poměrem velikostí nekomprimovaného (vstupního) a komprimovaného (výstupního) souboru. Tato veličina nemá žádnou jednotku, můžeme se tedy setkat s její hodnotou vyčíslenou například pomocí poměru či zlomku. Při kompresi souboru velkého 100 MB do souboru velkého 50 MB bychom dosáhli kompresního poměru 5/10, tedy 1:2. Výpočet kompresního poměru je znázorněn na rovnici číslo 1.

$$Kompresní\ poměr = \frac{Velikost\ komprimovaných\ dat}{Velikost\ nekomprimovaných\ dat} \tag{1}$$

Úspora místa je hodnota vyjadřující, kolik procent velikosti originálního souboru obsahuje zkomprimovaný soubor. Vypočítá se pomocí rovnice číslo 2 a vyjadřuje se v procentech. Na příkladu originálního 50 MB souboru a nového 35 MB souboru se úspora místa vypočítá jako $1 - (7/10) = 1 - 0,7 = 0,3 * 100 = 30\%$. Úspora dat je tedy 30%.

$$Úspora\ místa = 1 - Kompresní\ poměr \tag{2}$$

3.1 Základní kompresní principy

Tato kapitola je věnována popisu základních kompresních principů, které jsou součástí většiny složitejších kompresních nástrojů. Obecně nazýváme algoritmus či metodu provádějící kompresi či dekompresi *kompresor*. Ve všech níže popsaných algoritmech budeme pracovat se vstupní abecedou, kde na jeden znak připadá jeden byte, tudíž osm bitů. V této kapitole budou algoritmy popsány pomocí pseudokódů. Použité metody i třídy jsou popsány v kapitole 4.1.1.

Podle principu komprese poté dělíme kompresní algoritmy na:

- **Statistické algoritmy** Tyto algoritmy se vyznačují vytvářením statistiky počtu výskytů jednotlivých znaků. Při kompresi se poté jednotlivé znaky nahrazují počtem opakování tohoto znaku či se znaky s nejvyšší četností kódují pomocí nejmenší možné bitové reprezentace. Typickým příkladem tohoto statistické komprese je Huffmanovo kódování, které je popsáno v kapitole 3.1.2.

- **Slovníkové algoritmy** Tyto algoritmy pracují na bázi nahrazování řetězců indexy či odkazy a značkami na jejich předchozí výskyt. Při kompresi si vytváří slovník použitých slov, či používají vyrovnávací paměť. Typickým příkladem tohoto druhu komprese je algoritmus Lempel-Ziv 1977 popsaný v kapitole 3.1.3.

3.1.1 Run-Length Encoding

RLE je typ algoritmu pro bezeztrátovou kompresi dat, který funguje na principu kódování posloupností stejných znaků do dvojic - (délka posloupnosti, znak). Spadá do kategorie statistických kompresních algoritmů, jelikož počítá četnosti jednotlivých znaků. Kompresní poměr této metody je silně závislý na proudu vstupních dat a na konkrétních implementačních detailech této metody. Využívá se například při kompresi obrázků, které mají velké oblasti stejných znaků (například velká oblast černé barvy).

3.1.1.1 Komprese Existuje několik variant RLE. Základní verze kompresoru prochází vstupní soubor znak po znaku a kontroluje, jestli jsou znaky stejné. Pokud ano, zvýší hodnotu proměnné, která ukládá počet výskytů právě čteného znaku. Pokud ne, zapíše počet výskytů znaku a udávaný znak. Při jiné variantě tohoto algoritmu se zapisuje speciální znak, díky kterému lze rozpoznat, který znak je zkomprimovaný a který byl v originálním souboru. Tento znak však musí být zvolen tak, aby se nevyskytoval v komprimovaném souboru. Pokud tímto znakem bude např. "@", pak bude posloupnost "AAAA" zapsáná jako "@4A". Další implementace pak například zapisují počet opakování pouze tehdy, pokud by zkomprimovaný zápis byl menší, než ten původní, či zapisují pouze počet opakování. "A" tedy bude zapsáno jako "A" a ne "1A" či "AAAAAAAAAA" bude zapsáno jako "9A" místo "10A", kdy číslo 9 značí počet opakování

a nikoliv počet výskytů, který značí číslo 10. Tímto způsobem se ušetří jeden byte. Počet opakování pak lze zapsat buď v textové podobě nebo v binární podobě, případně můžeme použít nějakou další kompresní metodu, jako Huffmanovo kódování (více vic kapitola 3.1.2) či Fibonacciho kódování. [14]

3.1.1.2 Dekomprese Při základním principu dekomprese se prochází soubor znak po znaku, viz řádky 2-3 na Algoritmu číslo 1. První čtený znak bude vždy obsahovat počet výskytů (viz řádky 4 - 7), druhý pak opakovaný znak (viz řádky 9 - 12). U varianty se zapisovaným speciálním znakem budeme také číst soubor znak po znaku. Pokud však právě čtený znak obsahuje zvolený speciální symbol, jako je například "@" (viz řádky 14 - 16), znamená to, že další čtený znak bude představovat počet opakování. Další znak pap bude reprezentovat zakódovaný znak. Na řádku 16 je znázorněno, jak se zapisuje znak, který není nijak zakódovaný.

Algoritmus 1: Dekomprese RLE

```
1  compressionFlag ← false;
2  while vstupní soubor není celý přečtený do
3  |  readCharacter ← načteme znak ze vstupního souboru;
4  |  if compressionFlag = true then
5  |  |  repeatCount ← načteme 8 bitů vstupního souboru jako jedno číslo;
6  |  |  compressionFlag ← false;
7  |  |  afterNumber ← true;
8  |  else
9  |  |  if afterNumber = true then
10 |  |  |  Zapiš do výstupního souboru readCharacter celkem repeatCount krát;
11 |  |  |  afterNumber ← false;
12 |  |  |  break;
13 |  |  end
14 |  |  if readCharacter = '@' then
15 |  |  |  compressionFlag ← true;
16 |  |  else
17 |  |  |  zapiš do výstupního souboru readCharacter;
18 |  |  end
19 |  end
20 end
```

Příklad 3.1. *Uveďme si příklad dvou vstupních posloupností o délce 32 znaků **S1** a **S2**:*

***S1**: AAAAABBJJJJKGGGGGGGGGSSSSDWAAAAS*

***S2**: AAKSHBIJJWHSOKALSPPPJSBSKJKSNNAS*

Výše zmíněné posloupnosti S1 a S2 budou tedy zakódovány podle principu zápisu počtu výskytů a bez speciálního znaku jako S1' a S2' následujícím způsobem:

***S1'**: 5A2B4J1K9G4S1D1W4A1S - 20 znaků.*

***S2'**: 2A1K1S1H1B1I2J1W1H1S1O1K1A1L1S3P1J1S1B1S1K1J1K1S2N1A1S - 54 znaků.*
Kompresní poměr $S_1 = \frac{20}{32} = 0,63$.
Úspora místa $S_1 = 1 - 0,63 = 0.37 =$ *30%.*
Kompresní poměr $S_2 = \frac{54}{32} = 1,69$.
Úspora místa $S_2 = 1 - 1,69 = 0.37 =$ *-69% tudíž nárust dat o 69%.*

Z porovnání lze tedy s jistotou říci, že tento algoritmus je silně závislý na proudu vstupních dat.

Co se kompresního poměru týče, nejlepší variantou se jeví algoritmus, který nepoužívá žádný speciální znak pro označení počtu opakování a zapisuje pouze počet opakování. V této variantě však bude potřeba ošetřit případy, bude-li ve vstupním souboru sekvence čísel, jelikož tato čísla přečtě jako počet opakování místo samotného znaku. Pokud bychom mohli předpokládat, že vstupní soubor neobsahuje číslice, pak může v nejhorším případě nastat kompresní poměr 1:1 a to pokud soubor nebude obsahovat žádné stejné sousední znaky, či například pokud by soubor obsahoval pouze posloupnost dvojic stejných znaků. Při posloupnosti 2 stejných znaků bude sice sekvence "AA" zapsána jako "1A", k úspoře místa však nedojde. Při zapsání pouze opakování se tedy vyhneme zápisu "A" jako "1A", který by mohl zvýšit kompresní poměr až na 1:2.

3.1.2 Huffmanovo kódování

Tato kompresní metoda patří mezi statistické algoritmy. Algoritmus po průchodu souborem spočítá četnost výskytů znaků a vytvoří takzvaný Huffmanův strom. Blíže kořeni tohoto binárního stromu jsou umístěny znaky s nejvyšší četností výskytu. Prvkům blíže ke kořeni jsou přiděleny kratší kódy, naopak prvkům dále od kořene jsou přiděleny delší kódy. Aby bylo možné provést dekompresi, je nutné společně se zakódovanými symboly přenést i právě huffmanův strom. Za tímto účelem stačí přenést pouze tabulku četností, jejíž velikost je omezena velikostí vstupní abecedy. Pro soubory řádově větší, než velikost tabulky, je tento datový nárust zanedbatelný, uvažujemel-li vstupní abecedu o 256 znacích.

Existují dvě základní varianty této metody komprese:

- Při **statické** variantě se prochází soubor celkem dvakrát. Při prvním průchodu se vytvoří statistika počtu výskytu jednotlivých znaků. Při druhém projití se pomocí této statistiky vytvoří Huffmanův strom. V dalším popisu se budeme věnovat právě této variantě.

- **Dynamická** varianta prochází soubor pouze jednou, čímž oproti statické variantě šetří čas. Vytváření Huffmanova stromu a samotná komprese je tak prováděna zároveň. Jelikož se ale musí statistika četností neustále upravovat za běhu, je dosaženo horšího kompresního poměru oproti statické metodě a to především díky tomu, že kompresor nemá vytvořenou optimální statistiku četností už na začátku kódování.

Huffmanovo kódování se řadí mezi *prefixové kódy*. To jsou takové kódy, kde žádný symbol jeho kódové abecedy[1] není předponou jiného symbolu abecedy. Je-li nějaký symbol vstupní abecedy zakódován jako '1", žádný další znak nemůže začínat "1...". Dalším příkladem prefixového kódu jsou například mezinárodní předvolby pro telefonní čísla. Pokud by kódy nesplňovaly tuto vlastnost, nebylo by možné provést dekompresi.

3.1.2.1 Komprese Statická metoda, jak již bylo zmíněno, potřebuje dva průchody souborem. Při prvním průchodu se přečtou všechny znaky a vytvoří se tabulka četností pro každý znak vstupního souboru. Tento postup je popsán na Algoritmu číslo 2 mezi řádky 2 a 9. Podle této statistiky se poté vytvoří binární strom. Nejprve jsou vytvořeny listové uzly ohodnocené znakem a četností (viz řádky 10 - 15). Uzly s nejnižší četností jsou opakovaně spojovány tak, že je pro něj vytvořen nový společný předek ohodnocený součtem četností těchto dvou uzlů. Tento postup je opakován, dokud nevznikne celý strom, jak je popsáno na řádcích 16 - 23.

Jakmile je Huffmanův strom vytvořen, provedeme ohodnocení hran a to obvykle tak, že znaky výše ve stromu jsou reprezentovány menším počtem bitů, než znaky nacházející se níže ve stromu. Pro nejvíce se vyskytující znak tak může být použit pouze jeden bit. Kód pro určitý znak určíme tak, že půjdeme k prvku od kořene stromu. Nicméně ohodnocení hran může být libovolné, pokud dodržíme, že každá hrana od uzlu k jeho potomkům je hodnocena odlišně. Nejčastěji se však hrany označují takto: Pokud pro průchod ke znaku použijeme levou větev stromu, přiřadíme prvku prefix 0 (viz řádek 29). Pokud použijeme pravou větev, použijeme 1 (viz řádek 30). Tento postup je popsán na řádcích 24 - 38.

Při druhém průchodu souboru se všechny znaky zakódují pomocí bitů určených dle vytvořeného stromu. Pro potřebu dekomprese je však nutné do komprimovaného souboru zapsat ještě tento binární strom, aby bylo jasné, který bit reprezentuje který znak. Princip komprese je podrobněji vysvětlen na příkladu č.3.2:

Příklad 3.2. *Uvažujme vstupní proud **S1**: STROMY JSOU ZELENE*

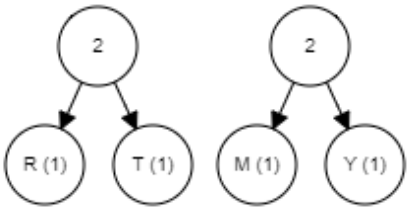
Počet výskytů znaků:												
S	*T*	*R*	*O*	*M*	*Y*	_	*J*	*U*	*Z*	*E*	*L*	*N*
2	*1*	*1*	*2*	*1*	*1*	*2*	*1*	*1*	*1*	*3*	*1*	*1*
Nejmenší četnosti spojíme do uzlů po dvou.												
S	~~*T*~~	~~*R*~~	*O*	~~*M*~~	~~*Y*~~	_	*J*	*U*	*Z*	*E*	*L*	*N*
2	~~*1*~~	~~*1*~~	*2*	~~*1*~~	~~*1*~~	*2*	*1*	*1*	*1*	*3*	*1*	*1*

Pro každý znak vytvoříme samostatný uzel. Dva uzly s nejnižší četností spojíme tak, že nově vzniklý uzel bude ohodnocen součtem četností spojovaným uzlů(viz Obrázek č. 1). Tento postup budeme opakovat, dokud nám nezbude pouze jeden uzel. Tento uzel reprezentuje právě kořen Huffmanova stromu.

[1]Vstupní posloupnoust znaků, v tomto případě znaky mohou nabývat pouze hodnot "1" či "0".

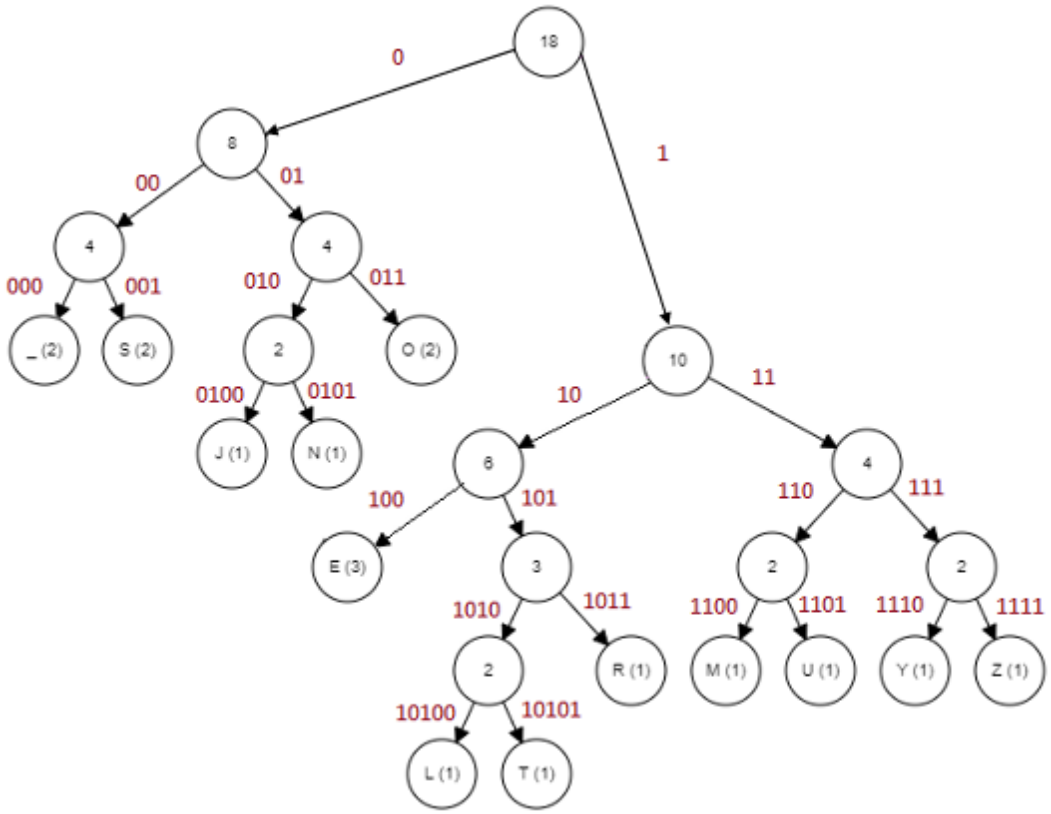
Algoritmus 2: Komprese pomocí Huffmanova kódování

```
1  Inicializujeme zásobník uzlů Heap;
2  while vstupní soubor obsahuje znaky do
3  |  readCharacter ← načteme znak ze vstupního souboru;
4  |  if readCharacter existuje v tabulce četností then
5  |  |  zvýšíme četnost prvku readCharacter
6  |  else
7  |  |  přidáme readCharacter do tabulky četností s četností ← 1
8  |  end
9  end
10 for iterujeme tabulkou četností do
11 |  Vytvoř uzel n;
12 |  n.Frequency ← FrequencyTable.Frequency;
13 |  n.character ← FrequencyTable.character;
14 |  Vlož n do Heap;
15 end
16 while Heap obsahuje uzly do
17 |  vytvoř Huffmanův strom root;
18 |  root.left ← Heap.node s nejmenší frekvencí;
19 |  Odeber prvek s nejmenší frekvencí z Heap;
20 |  root.right ← Heap.node s nejmenší frekvencí;
21 |  Odeber prvek s nejmenší frekvencí z Heap;
22 |  Vlož root do Heap;
23 end
24 Vytvoř frontu Queue;
25 Vytvoř Huffmanův strom node a vlož jej do Queue;
26 while Queue není prázdná do
27 |  Huffmanův strom left ← levý potomek;
28 |  Huffmanův strom right ← pravý potomek;
29 |  left.code+ = 0;
30 |  right.code+ = 1;
31 |  if left.left obsahuje instanci Huffmanova stromu then
32 |  |  Vlož left a right do Queue;
33 |  |  Smaž node z Queue;
34 |  else
35 |  |  Tabulka četností v místě [left.character].code ← left.code;
36 |  |  Tabulka četností v místě [right.character].code ← right.code;
37 |  |  Smaž left a right z Queue;
38 |  end
39 end
40 while vstupní soubor obsahuje znaky do
41 |  readCharacter ←přečteme znak vstupního souboru;
42 |  for iterujeme tabulkou četností do
43 |  |  if tabulka četností.Character = readCharacter then
44 |  |  |  Result ←tabulka četností.code ;
45 |  |  |  break;
46 |  |  end
47 |  end
48 end
49 zapíšeme Result do výstupního souboru;
```



Obrázek 1: Spojení 2 znaků s nejmenší četností do uzlu se součtem jejich četností.

Binární kód se poté určí tak, že půjdeme od kořene až k hledanému listu. Cestě vlevo vždy přiřadíme 0 a cestě vpravo vždy 1. Na Obrázku č. 2 se nachází ukázka celého Huffmanova stromu včetně ohodnocení vrcholů.



Obrázek 2: Huffmanův strom s ohodnocením vrcholů a hran.

*Například znak E s nejvyšší četností výskytu tedy bude zakódován pomocí **100**.*

*Věta **S1** bude tedy zakódována takto:*

***S1**: STROMY JSOU ZELENE - 18 bytů*

Zakódováno pomocí ascii:

***S1**: 010100110101010001010010010011110100110101011001001000000100101001010011010011101010101001000000101101001000101010011000100010101001110010001 01 = 144bitů.*

Po kompresi:

***S1'**: 00110101101101111001110000010000101111010001111100101001000101100 - 65 bitů*

Z porovnání lze tedy vidět, že z původních 144 bitů jsme se po použití Huffmanova kódování dostali na 65 bitů, což odpovídá 45% původního souboru. Úspora místa tak činí 55% Do souboru však budeme muset ještě zapsat reprezentaci Huffmanova stromu pro potřebu dekomprese, což však u rozsáhlejších souborů kompresní poměr výrazně nezhorší.

3.1.2.2 Dekomprese Při dekódování se nejprve přečtou instrukce pro vytvoření Huffmanova stromu, tedy například tabulka četností, ze které lze Huffmanův strom rekonstruovat. Dále se ve vstupním souboru čte bit po bitu a využívá se právě již vytvořeného stromu. Právě čtený bit udržuje informaci o tom, na kterého potomka uzlu budeme postupovat. Začínáme vždy od kořene stromu a postupujeme směrem k listovým uzlům. Pokud je po přečtení bitu a postupu v hierarchii stromu možnost jít ve stromu dále, znamená to, že jsme nepřečetli celý znak. Přečte se tedy další bit a takto se bude pokračovat do té doby, než se budeme nacházet v uzlu, který nemá žádného dalšího potomka. Názorně je tento postup dekomprese vysvětlen na Příkladu číslo 3.3.

Příklad 3.3. *Uvažujme zakódovanou sekvenci **S1**: 0010110101*

Uvažujme Huffmanův strom, který nalezmeme na Obrázku číslo 2.

Přečteme 0 a ve stromu se posuneme doleva. Jelikož z uzlu s číslem 8 vede další cesta, přečteme další znak. Takto budeme číst znak po znaku, dokud se nedostaneme do uzlu, ze kterého nevede žádná další cesta. Znak, který je uložen v tomto uzlu je znak zakódovaný pomocí kódu, který jsme právě přečetli. Takto budeme pokračovat do té doby, dokud nebude počet dekódovaných znaků stejný, jako celkový počet znaků ze vstupní tabulky četností.

*Pokud se podíváme hrany stromu na Obrázku číslo 2, zjistíme, že po přečtení sekvence **001** dojdeme k uzlu S, ze kterého nevede žádná další cesta. **001** tedy reprezentuje písmeno S. Při čtení dalšího bitu 0 budeme opět postupovat od kořene stromu.*

Kód	*Znak*
001	*S*
011	*O*
0101	*N*

*Sekvence **0010110101** tedy bude odpovídat slovu **SON**.*

3.1.3 LZ77

Bezeztrátový kompresní algoritmus Lempel-Ziv 1977 patří mezi takzvané slovníkové algoritmy. Jelikož je tento algoritmus velice úspešný, existuje velké množství jeho derivátů. Algoritmus prochází soubor od začátku do konce a v případě, že narazí na opakující se skupinu znaků,

nahradí tuto skupinu ukazatelem na její předchozí výskyt. Ukazatel je zakódován jako vzdálenost od současné pozice, délka posloupnosti znaků a následující znak. Existuje i verze [7] tohoto algoritmu, kde se odkaz uloží pouze jako <Vzdálenost od předchozího výskytu, délka sekvence>. K úspoře místa tak dochází, je-li odkaz kratší než kódováná posloupnost znaků.

3.1.3.1 Komprese Komprese probíhá v jednom čtení souboru. Při průchodu souborem se využívají 2 pole: Vyhledávací pole (anglicky search buffer, dále jen *SB*) a dopředné pole (anglicky look-ahead buffer, dále jen *LAB*). Na Algoritmu číslo 3 je vysvětlen princip nalezení předchozího výskytu řetězce a následného zakódování. V *SB* se ukládají znaky, které už byly přečteny (řádek 5). V *LAB* se nacházejí znaky, které jsou právě čteny nebo brzy budou čteny (řádky 6 - 9). První znak *LAB* se porovnává se znaky v *SB* a to od konce po začátek (řádek 13). Při nalezení shodného znaku je čten další znak (řádky 11, 14 - 16). Tyto dva znaky jsou už poté hledány společně (řádek 12). Při vyhledávání tohoto řetězce v *SB* se jde zpětně k začátku souboru. Při nalezení sekvence se do řetězce přidává další znak z *LAB* tak dlouho, dokud lze tento řetězec najít v *SB* (řádky 11 - 16). Nejdelší možný řetězec se poté zakóduje jako (Vzdálenost od sekvence v SB, délka řetězce, následující znak) (řádky 18 - 21). Jelikož mají oba buffery určitou délku, při každém dalším čtení znaku se posouvají (řádky 24 - 27).

3.1.3.2 Dekomprese Při dekompresi přečteme 3 znaky, jak je znázorněno na Algoritmu číslo 4 mezi řádky 1 až 4. První znak obsahuje informaci o vzdálenosti sekvence od předchozího výskytu i. V následujícím znaku j se poté nachází číslo označující, jak dlouhá sekvence byla. V již dekódovaném textu se tedy vrátíme o i znaků a zapíšeme na jeho konec j znaků (viz řádek 6). Nakonec ještě přidáme znak, který se nachází ve třetím čteném znaku x (viz řádky 8 - 9). Pokud se v druhém čteném znaku j nachází 0, do výstupního souboru dáme pouze poslední čtený znak x (viz řádky 5,8,9). Znamená to totiž, že tento znak nebyl při dekompresi nalezen ve vyhledávacím okně.

Příklad 3.4. *Ve vstupním souboru se nachází řetězec* ***nana_leje_na_koleje****, který budeme komprimovat. Délku SB a LAB máme nastavenou na 10 znaků. Postup je znázorněn v Tabulce č. 1:*

V prvním kroku zapíšeme (0,0,n), jelikož znak n nebyl nalezen v SB. První nula zde odpovídá vzdálenosti od předchozího výskytu. Druhá nula odpovídá délce zakódovaného řetězce. Znak n je poté následující řetězec za sekvencí popsanou předchozími dvěma čísly. Jelikož je však sekvence 0,0, tudíž prázdná, uloží se do posledního pole tento hledaný znak.

V tomto případě však nedošlo k úspěšné kompresi souboru, jelikož kompresní poměr je větší než jedna. Z původního 19 B souboru se po kompresi stal 30 B soubor a to hlavně proto, že znaky, které nebyly nalezeny, jsou zakódovány pomocí 3 B oproti originálnímu 1 B.

Algoritmus 3: Nalezení předchozího výskytu řetězce pomocí LZ77

```
1  maxVelikostSB ← 255;
2  delka ← 0;
3  pozice ← 0;
4  nalezenaPozice ← 0;
5  Inicializujeme pole znaků SB;
6  Inicializujeme pole znaků LAB;
7  while Vstupní soubor obsahuje znaky do
8  |  LAB = LAB+ načteme znak ze vstupního souboru;
9  end
10 for i ← 0; i <délka LAB; i++ do
11 |  for j ← 0; j < maxVelikostSB; j++ do
12 |  |  hledanyRetezec ← j znaků z LAB od pozice i;
13 |  |  nalezenaPozice ← první výskyt hledanyRetezec v SB;
14 |  |  if Znak byl nalezen then
15 |  |  |  delka ← j;
16 |  |  |  pozice ← nalezenaPozice;
17 |  |  else
18 |  |  |  Zapiš do výchozího souboru pozice v binární podobě;
19 |  |  |  Zapiš do výchozího souboru delka − 1 v binární podobě;
20 |  |  |  Zapiš do výchozího souboru znak LAB na pozici j;
21 |  |  |  Ukonči cyklus;
22 |  |  end
23 |  end
24 |  Zvětši i o delka;
25 |  Odeber prvních delka prvků z SB;
26 |  Přidej delka znaků z LAB do SB;
27 |  Odeber prvních delka prvků z LAB;
28 |  delka ← 0;
29 |  pozice ← 0;
30 |  nalezenaPozice ← 0;
31 end
```

Algoritmus 4: Dekomprese podle LZ77

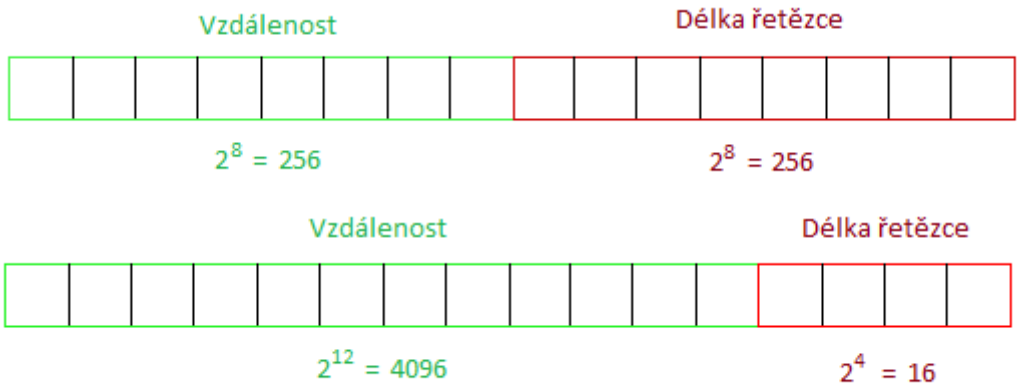
```
1  while vstupní soubor obsahuje znaky do
2  |  i ← načteme 8 bitů vstupního souboru jako jedno číslo;
3  |  j ← načteme 8 bitů vstupního souboru jako jedno číslo;
4  |  x ← přečteme vstupní znak;
5  |  if j > 0 then
6  |  |  dekodovanyText ←
   |  |    dekodovanyText.SubString(délka dekódovaného textu − i, j)
7  |  end
8  |  if x obsahuje znak then
9  |  |  dekodovanyText += x;
10 |  end
11 end
```

Tabulka 1: Postup kódování vstupní sekvence pomocí LZ77.

SB	LAB	Výstup	Posun bufferu
	nana_leje_na_koleje	(0,0,n)	1
n	ana_leje_na_koleje	(0,0,a)	1
na	na_leje_na_koleje	(2,2,_)	3
nana_	leje_na_koleje	(0,0,l)	1
nana_l	eje_na_koleje	(0,0,e)	1
nana_le	je_na_koleje	(0,0,j)	1
nana_lej	e_na_koleje	(2,1,_)	2
na_leje_	na_koleje	(8,3,k)	4
eje_na_k	oleje	(0,0,o)	1
je_na_ko	leje	(10,3,e)	4

Tento problém se dá obvykle vyřešit následným použitím Huffmanova kódování(viz kapitola č. 3.1.2), který může například nuly zakódovat efektivněji. Komprese LZ77 by se projevila u delšího textu, kde se například budou opakovat celá slova.

Pro zápis vzdálenosti a délky předchozího výskytu jako číselných hodnot jsou využity 2 B, ale do 1 B lze zapsat pouze 256 číselných hodnot. Jelikož většina slov evropských jazyků není delší než 16 znaků, některé algoritmy odvozené z LZ77 mouhou využít větší počet bitů pro zápis vzdálenosti a menší pro zápis řetězce, jak je znázorněno na obrázku číslo 3.



Obrázek 3: Rozdělení 2 Bytů pro lepší rozdělení vyrovnávací paměti popsané podle [7].

3.1.4 Move-To-Front Transform

Nejedná se přímo o typ komprese, ale o typ transformace, což znamená, že výsledný soubor bude možné efektivněji komprimovat pomocí jiné kompresní techniky. Znaky vstupního proudu jsou kódovány jako indexy těchto znaků z tabulky znaků. Nejprve je třeba vytvořit tabulku indexů, která začíná indexem 0. Tato tabulka je předem definována a je stejná pro kompresori i dekompresi. Oproti Huffmanovu kódování se tak v této tabulce nacházejí všechny znaky abecedy a ne jen ty, které se vyskytují v textu. Na druhou stranu je tabulka předem daná a nemusí se tak zapisovat do komprimovaného souboru, tudíž nezabírá místo. Celý soubor je poté kódován pomocí indexu znaku z této tabulky v binární podobě.

3.1.4.1 Komprese Při čtení každého znaku se nahlíží do tabulky znaků, kterou si lze představit jako pole znaků, kdy k jednotlivým prvkům pole můžeme přistupovat pomocí jejich indexu. (viz Algoritmus číslo 5, řádek 7). Na řádcích 2 - 3 je vidět naplnění tabulky pomocí znaků s ascii hodnotou 0 - 255. Přečtený znak je poté zakódován pomocí indexu tohoto znaku z tabulky (viz řádky 6 - 8). V tabulce je poté právě použitý znak přesunut na začátek tabulky (viz řádky 9 -

14), čímž je mu přidělen index 0. Odtud také pochází název „Move to front“, neboli „přesunout dopředu“. Pokud se tedy znak opakuje několikrát za sebou, je zakódován sekvencí nul, což je vhodné například pro RLE, které je popsáno v kapitole 3.1.1.

3.1.4.2 Dekomprese Při dekompresi se postupuje stejně jako u komprese s rozdílem, že čteme číselné hodnoty ze souboru a přiřazujeme jim znakové hodnoty z tabulky. Při použití znaku je opět tento znak přesunut na začátek tabulky a má tak index 0.

Algoritmus 5: Komprese MTF

```
1  inicializujeme pole znaků MapOfCharacters o velikosti 255 typu char;
2  for int i = 0; i<255;i++ do
3  |  MapOfCharacters[i] = i;
4  end
5  while vstupní soubor není prázdný do
6  |  readCharacter ← přečteme vstupní znak;
7  |  CharAsNumber ← MapOfCharacters[ReadCharacter];
8  |  zapíšeme CharAsNumber do výstupního souboru jako číslo v binární podobě;
9  |  t = MapOfCharacters[CharAsNumber];
10 |  for i ← CharAsNumber − 1 ; i > 0 ; i − − do
11 |  |  nextSymbol ← MapOfCharacters[i + 1];
12 |  |  MapOfCharacters[i + 1] ← MapOfCharacters[i];
13 |  end
14 |  MapOfCharacters[0] ← t;
15 end
```

Příklad 3.5. *Předveďme si transformaci MTF na následujím příkladu: Budeme pracovat s tabulkou indexů pro abecedu A - Z:*

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
znak	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*	*M*
index	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*
znak	*N*	*O*	*P*	*Q*	*R*	*S*	*T*	*U*	*V*	*W*	*X*	*Y*	*Z*

*Kódovat budeme sekvekci **S1**: PAPAT*
„P“ tedy zakódujeme jako číslo 15, jelikož se nachází jako 16. znak v mapě (mapa začíná indexem 0). V mapě se poté písmeno P přesune na začátek. (PABCDEFGHIJKLMNOQRSTUVWXYZ). Zbytek kódování probíhá následovně:

Slovo	***Zakódované***	***Mapa***
P*APAT*	*15*	*(ABCDEFGHIJKLMNO**P**QRSTUVWXYZ)*
*P**A**PAT*	*15,1*	*(P**A**BCDEFGHIJKLMNOQRSTUVWXYZ)*

*PA**P**AT 15,1,1 (A**P**BCDEFGHIJKLMNOQRSTUVWXYZ)*
*PAP**A**T 15,1,1,1 (P**A**BCDEFGHIJKLMNOQRSTUVWXYZ)*
*PAPA**T** 15,1,1,1,19 (APBCDEFGHIJKLMNOQRS**T**UVWXYZ)*

*Výsledná sekvence **S1'** tedy bude vypadat: 15 1 1 1 19*
Z porovnání je tedy vidět, že se velikost souboru nijak nezmění. Z výsledku lze pak dále vidět, že by tato posloupnost byla lépe zkomprimována pomocí Huffmanova kódování či RLE, viz kapitoly 3.1.2 a 3.1.1. RLE by v tomto případě využilo posloupnosti "111".

3.1.5 Aritmetické kódování

Aritmetické kódování je metoda pro bezeztrátovou kompresi dat řadící se mezi statistické metody. Aritmetické kódování kóduje všechny symboly do jediného čísla, narozdíl od Huffmanova kódování, které přiřazuje každému znaku určitou hodnotu. Navíc lze dosáhnout lepšího kompresního poměru, než je 1 bit na 1 znak, jak je tomu u Huffmanova kódování. Algoritmus rozděluje znaky podle počtu výskytů do určitých intervalů a s nimi poté při svých výpočtech pracuje. Problémem u tohoto druhu komprese je však přesnost, jelikož se zvyšujícím se počtem kódovaných znaků se zmenšuje interval a pro jeho správné určení se zvyšují nároky na přesnost tohoto čísla. Tento problém se dá řešit tak, že se do výsledného souboru zapíše ta část intervalu, která se při výpočtech nemění. Například pokud se interval pohybuje mezi 0.12 a 0.14, do výstupního souboru lze zapsat číslo 0.1.

Obdobou aritmetického kódování je rozsahové kódování. Rozdíl je v tom, že aritmetické kódování ukládá výsledné číslo pouze v bitové podobě [17], zatímco rozsahové může zapisovat číslo v jakékoliv soustavě(například desítkové či šestnáctkové).

3.1.5.1 Komprese Jak již bylo zmíněno, aritmetické kódování patří mezi statistické metody. Díky tomu je třeba celý soubor nejprve přečíst a vytvořit statistiku četností pro jednotlivé znaky. Zvláštností u toho druhu komprese je, že do této tabulky ukládá i EoF[2] znak. Poté se určí rozsah, ve kterém se vypočítá výsledné číslo (například interval <0, 1>). Tento interval se rozdělí na podintervaly podle četností. Pokud je tedy v souboru o 100 znacích některý symbol 6 krát, bude jeho interval zabírat právě 6% celkového intervalu. Při kódování znaku se vybere ten interval, do kterého tento právě kódovaný znak spadá. V dalším kroku se tento podinterval stane novým celým intervalem a ten se následně opět rozdělí podle četnosti znaků na podintervaly jako v předchozím kroku. Takto se kóduje celá zpráva včetně znaku pro konec souboru. Do výsledného souboru se poté kromě výsledného čísla zapíše i statistika četností. Výsledné číslo představuje libovolné číslo spadající do výsledného intervalu. Existuje i možnost zapsat na začátek souboru velikost originálního souboru, díky které budeme vědět, kdy skončit i bez použití znaku pro konec souboru. Příklad komprese je popsán na Příkladu číslo 3.6.

[2]Označení pro konec souboru(End of file).

Princip zmenšování intervalů je popsán na Algoritmu číslo 6. Na řádcích 1 a 2 je určena horní a spodní hranice počátečního intervalu. Na řádku číslo 3 je definován cyklus, který prochází celý vstupní soubor. Na řádku číslo 4 je přiřazena dvojice čísel do proměnné *val*, která reprezentuje procentuální hodnotu celkového intervalu, pro právě čtený znak. Na řádcích 5 a 6 je poté tato procentuální hodnota převedena na skutečné číslo z intervalu, se kterým algoritmus právě pracuje. Na řádcích 7 a 8 je tento interval zmenšen pro další krok cyklu.

Algoritmus 6: Algoritmus na zmenšování intervalů až na výsledný interval.

```
1 range_start ←Spodní hranice intervalu;
2 range_end ←Horní hranice intervalu;
3 for i ← 0; i < velikost vstupního souboru; i++ do
4 |   value ← range[ přečteme další znak vstupního souboru];
5 |   range_start1 =
  |     ((value.StartOfInterval ∗ (range_end − range_start))/100) + range_start;
6 |   range_end1 =
  |     ((value.EndOfInterval ∗ (range_end − range_start))/100) + range_start;
7 |   range_start = range_start1;
8 |   range_end = range_end1;
9 end
```

3.1.5.2 Dekomprese Dekomprese probíhá podobným způsobem jako komprese. Nejprve se z komprimovaného souboru přečte statistika četností znaků a pomocí nich se vytvoří příslušné podintervaly. Poté se ze souboru přečte číslo, pomocí kterého byl soubor zakódován. Zakódovaný znak je dekódován pomocí intervalu, do kterého číslo spadá a který patří tomuto znaku. Intervaly se v každém dalším kroku zmenšují do té doby, než bude číslo spadat do intervalu, který přísluší znaku pro konec souboru.

Příklad 3.6. *Pro příklad máme vstupní soubor **S1**: AAB. Po přečtení souboru dostane následující četnosti:*

***A** : 2, **B** : 1, & : 1, kdy & je EoF znak.*

Budeme pracovat s intervalem <0,1> a znakům přídělíme intervaly:

***A**<0,0.5>; **B**<0.5,0.75>; &<0.75, 1>.*

Následující postup je ilustrován na Obrázku číslo 4. Kódujeme soubor zleva znak po znaku. V první iteraci tedy kódujeme znak "A". Ten spadá do intervalu <0,0.5>. Pro další iteraci se omezíme na interval <0,0.5>, který opět rozdělíme v poměrech podle četností. Znaky teď budou mít intervaly:

***A**<0, 0.25>; **B**<0.25,0.375>; &<0.375, 0.5>.*

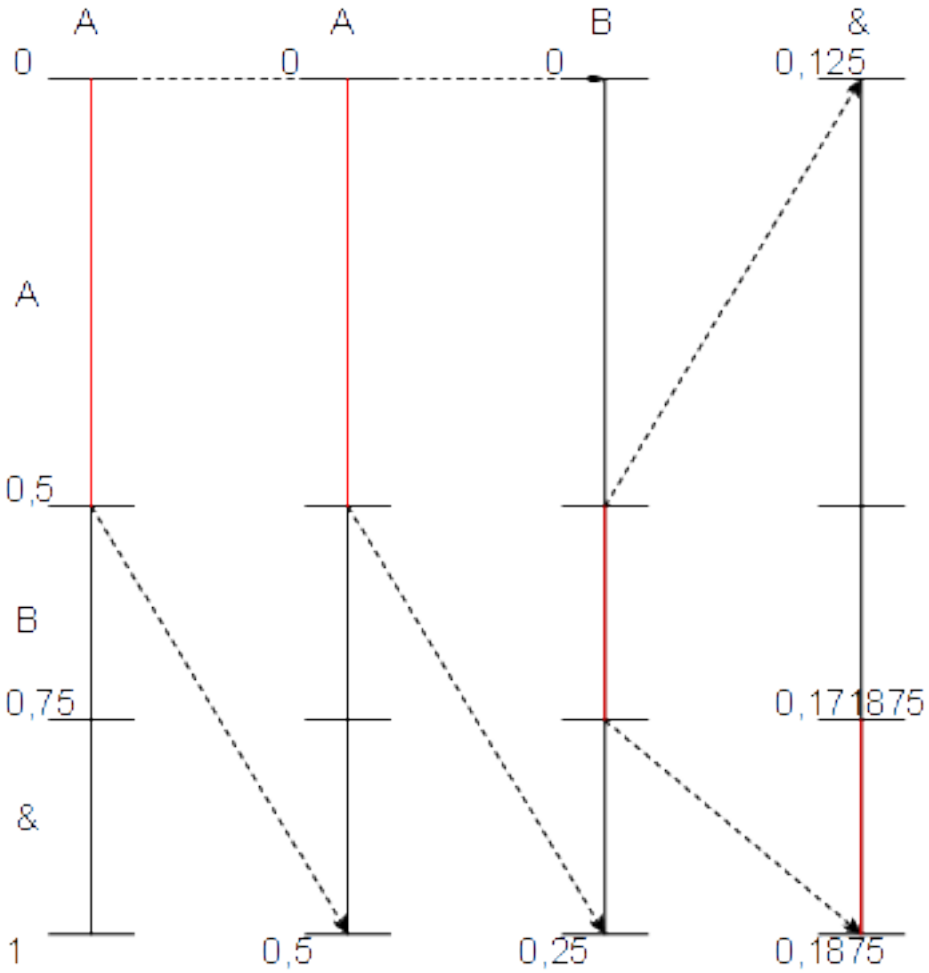
Nyní kódujeme opět znak "A", náš další interval tedy bude v rozsahu <0,0.25>. Tento interval opět rozdělíme na podintervaly:

***A**<0,0.125>; **B**<0.125,0.1875>; &<0.1875, 0.25>.*

Jelikož je kódovaný znak "B", který náleží do intervalu <0.125,0.1875>, pro další iteraci vybereme právě tento interval. Ten se však opět, tentokrát naposledy, jelikož nyní budeme kódovat EoF znak &, zmenší na intervaly:

***A**<0.125,0.15625>; **B**<0.15625,0.171875>; **&**<0.171875,0.1875>;*

Výsledný interval je tedy <0.171875,0.1875>, ze kterého vybereme libovolné číslo a to zapíšeme jako výsledek do souboru. Takové číslo je například 0.18, jak je znázorněno na obrázku č. 4.



Obrázek 4: Grafické znázornění zmenšování intervalů pomocí Aritmetického kódování.

3.1.6 Burrows-Wheeler Transform

Je pomocný algoritmus využívaný při kompresi dat, který nijak nemění hodnotu žádného ze znaků, pouze mění jejich pořadí. Algoritmus funguje na tom principu, že vytváří rotace vstupního řetězce. Mezi těmito znaky se nachází i znak EoF. Tyto rotace se nakonec seřadí a z každé rotace se vybere poslední znak. Díky seřazení dat se výstupním souboru budou vyskytovat delší

sekvence stejných znaků. Toto je výhodné například pro RLE, kterému se věnuje kapitola 3.1.1. Jak tento algoritmus funguje je názorně ukázáno na Příkladu číslo 3.7.

3.1.6.1 Komprese Pro kompresi dat se vytvoří všechny rotace vstupního řetězce včetně symbolu označujícího konec souboru (EoF). Ty se poté abecedně seřadí a jako výstup se použije vždy poslední znak každé z těchto rotací.

3.1.6.2 Dekomprese Při dekompresi uvažujeme vstupní soubor jako pole. Vytvoříme tabulku, do které se vždy vloží jako sloupec celý vstupní soubor a abecedně se seřadí. Takto se v každém dalším kroku vkládají všechny znaky vstupního souboru před první sloupec a pokaždé se znovu seřadí až do té doby, než tabulka bude mít tvar čtvercové matice. Jako výstup se poté vybere ten řádek tabulky, který končí znakem EOF.

Příklad 3.7. *V tomto příkladu si popíšeme, jak se zakóduje slovo „BANNANNA“. Jako vstupní řetězec algoritmu se však použije řetězec S1: BANNANNA@*
Znak @ označuje konec souboru.

Vstup	***Rotace***	***Seřazené***	***Výstup***
	BANNANNA@	*ANNANNA@**B***	
	@BANNANNA	*ANNA@BAN**N***	
	A@BANNANN	*A@BANNAN**N***	
	NA@BANNAN	*BANNANNA@*	
BANNANNA@	*NNA@BANNA*	*NANNA@BA**N***	*BNN@NNAAA*
	ANNA@BANN	*NA@BANNA**N***	
	NANNA@BAN	*NNANNA@B**A***	
	NNANNA@BA	*NNA@BANN**A***	
	ANNANNA@B	*@BANNANN**A***	

Výstupem komprese tedy bude sekvence S1': BNN@NNAAA
Jak je z výsledku vidět, znaky se nyní za sebou opakují, což je opět vhodné například pro RLE, popsané v kapitole 3.1.1.

3.2 Kompresní nástroje bez ohledu na strukturu XML

Jelikož je XML soubor vlastně textový dokument, lze pro jeho kompresi úspěšně využít většinu algoritmů specializovaných na kompresi textu. Výhoda tohoto pohledu na kompresi je v množství existujících kompresních nástrojů a jejich jednoduchém využití. Některé z nich jsou i velmi účinné. Kompresní nástroje popsané v této kapitole jsou ve většině případů kombinací základních kompresních principů popsaných v kapitole 3.1.

3.2.1 GZip

Tato metoda bezeztrátové komprese dat je založena na kompresním algoritmu Deflate a využívá pro kompresi kombinaci LZ77 (viz kapitola 3.1.3) a Huffmanovo kódování (viz kapitola 3.1.2). Samotná komprese probíhá ve dvou bodech. Nejprve se odstraní opakující se řetězce pomocí algoritmu LZ77 a poté se tyto značky<pozice,délka,další znak> rozdělí na 3 proudy(pozice, délka a znak), které se zvlášť komprimují pomocí Huffmanova kódování. GZip je jeden z nejznámějších a nejrychlejších kompresních nástrojů. O tomto vypovídá i fakt, že tento kompresní nástroj je implementován v mnoha vývojových prostředích a operačních systémech (Soubor *.zip). Všeobecně pak existuje zažitý pojem "Zazipovat soubor", který vypovídá právě o kompresi dat pomocí tohoto nástroje.

3.2.2 BZip2

Tato metoda bezeztrátové komprese dat využívá Burrows-Wheeler Transform, Move-To-Front transform, Run-Length encoding a Huffmanovo kódování, přičemž komprimuje bloky dat o velikosti 100-900 kB (Nastavitelné po 100kB). Dosahuje lepších výsledků než algoritmus Deflate co se úspory místa týče, na druhou stranu je více časově náročná. První verze BZip využívala Aritmetické kódování, to však bylo nahrazeno Huffmanovým kódováním z důvodu existence patentu na Aritmetické kódování [2]. Tento princip bohužel neumí komprimovat více souborů zároveň, je proto nutné využít některý z jiných nástrojů, který více souborů nejprve spojí v jeden a až poté nastupuje BZip2.

3.2.3 PPM

Prediction by Partial Matching je adaptivní statická kompresní metoda založená na modelech kontextu a předpovědi znaků. Modely ukládají předchozí znaky a pomocí nich v nekomprimované části předpovídají výskyt znaků. Daný symbol je poté pomocí dané pravděpodobnosti zakódován pomocí aritmetického kódování (popsáno v 3.1.5). Tento princip se využívá především ve shlukové analýze[3]. Vznik této metody sahá až do 90. let minulého století, avšak její rozšíření se v té době setkávalo s problémy v podobě nedostatečné operační paměti. Tato metoda je totiž jak

[3]Shlukování jednotek do skupin na základě podobnosti.

časově, tak především paměťově náročná, jelikož si ukládá kontextové modely, což jsou tabulky, které nám říkají, jaké znaky a kolikrát již byly použity.

Pracuje se s několika modely zároveň a každý z nich má určeno, kolik znaků si bude pamatovat. Podle počtu těchto znaků n se určuje stupeň PPM a zápis této metody se poté značí pomocí PPMn. Pokud model není nijak omezen maximálním počtem uchovávaných znaků, tato metoda se značí jako PPM*. Při výpočtu předpovědi výskytů následujících znaků se nahlíží do kontextových modelů. Vypočtená pravděpodobnost je využita k zakódování znaků pomocí aritmetického kódování (viz kapitola č. 3.1.5).

Výpočet předpovědi probíhá tak, že se právě čtený znak porovnává se znaky v modelech. Pokud je znak nalezen v kontextovém modelu s nejvíce znaky, je pravděpodobnost určena jako četnost výskytu znaku v daném modelu. Pokud není znak nalezen, prohledávají se zbylé modely. Pokud ani tam není znak nalezen, předpověď se určuje pomocí přednastavené hodnoty. Modely jsou pak upraveny tak, aby byl opět zachycen právě zakódovaný znak.

Existuje několik variant PPM, kdy každá z nich řeší určování pravděpodobností různě.

3.2.4 LZMA

Lempel-Ziv-Markov-Chain Algorithm je bezeztrátová kompresní metoda založena na vylepšené verzi LZ77 (viz kapitola 3.1.3). K tomu jsou navíc využívány Markovovy řetězce[4] [13] a rozsahové kódování (viz kapitola 3.1.5). Tato metoda je hlavní metodou používanou v nástroji 7-ZIP produkujím soubory ve formátu *.7z. Metoda využívá stejně jako LZ77 slovník, který je však oproti ostatním kompresním slovníkovým metodám několikrát větší a jeho velikost je uživatelsky nastavitelná až do 4 GB. Díky tomu tato metoda vyžaduje mnohem větší množství paměti, než ostatní nástroje a její komprese je relativně pomalá. Naopak dekomprese je několikanásobně rychlejší, než komprese. Celkově algoritmus LZMA dosahuje ve většině případů výrazně lepších výsledků komprese dat oproti LZ77.

3.3 Kompresní nástroje s ohledem na strukturu XML

Oproti kompresi XML jako textu se tyto kompresní algoritmy zaměřují na vnitřní strukturu XML dokumentů a využívají jejich sémantické informace. Pomocí nich dále upravují XML data k další fázi procesu, ve které většinou nastupují již zmíněné běžné kompresní algoritmy. Struktura XML dokumentů se skládá z více typů uzlů (elementy, atributy, komentáře, text, anotace, atd.), avšak pro zjednodušení uvažujeme pouze elementy a atributy.

Existují kompresory, které ignorují bílé znaky, které obvykle nemají v XML žádný význam. Dekomprimovaný soubor pak nemusí být 100% bitově shodný s originálním souborem, ale obsažená informace se dá chápat jako shodná.

[4] Algoritmus vyvinut pro matematické statické modelování, který původně nijak nesouvisel s kompresí dat.

Kompresory pracující se strukturou XML se dělí do dvou základních kategorií:

- S podporou dotazování.
- Bez podpory dotazování.

3.3.1 XML komprese s podporou dotazování

Hlavním cílem tohoto druhu komprese XML dat je možnost práce s daty i po kompresi souboru. Není tedy třeba data dekomprimovat, aby v souboru mohly být vyhodnoceny dotazy, jako jsou například dotazy XPath[5]. Díky tomu je však kompresní poměr zpravidla horší, než u komprese bez podpory dotazování. Tento druh komprese se používá například při práci s velkými soubory, jelikož analýza celého nekomprimovaného souboru může být časově náročnější, než analýza určité části komprimovaných dat.

3.3.1.1 XGrind Je jeden z prvních XML kompresorů s podporou dotazování. Jedná se o takzvaný homomorfní kompresor [18], což znamená, že zachovává strukturu původního XML souboru a je tak možné přistupovat ke komprimovanému souboru stejně jako k souboru nekomprimovanému. XGrind při vytváření struktury dokumentu využívá DTD[6]. XGrind implementuje sémantické kompresory, což jsou kompresní algoritmy, které komprimují sémanticky podobná data stejným principem a díky tomu tak mohou nástroje, které je implementují, dosáhnout ještě lepšího výsledku komprese. XGrind rozděluje vstupní data na 3 typy:

- **Struktura** Komprese struktury funguje na principu slovníkového kódování. Elementy označuje jako **T** a atributy jako **A**, za těmito identifikátory pak následuje unikátní kód, který odkazuje do slovníku, kde se nacházejí originální názvy těchto elementů a atributů. Koncové značky se poté kódují pouze pomocí symbolu /, jelikož se při dekompresi obnoví pomocí počáteční značky.
- **Výčtové hodnoty dat atributů** Jelikož se v XML souborech často vyskytují výčtové typy jako hodnoty atributů, jako jsou poštovní směrovací čísla či názvy států, s pomocí DTD XGrind tyto typy automaticky zjistí a zakóduje je pomocí odkazu do slovníku. Tato data jsou komprimována sémantickým kompresorem označeným jako „enum“. Ten nahrazuje data výčtových typů číslem, které určuje jejich index v tabulce výčtových typů.
- **Ostatní hodnoty atributů a elementů** Komprese obsahu se provádí pomocí sémantického kompresoru „nahuff“. Ten provádí kompresi pomocí bezkontextové komprese, jelikož si XGrind zakládá na možnosti dotazovat se nad souborem bez nutnosti dekomprese. Ta přiřazuje identifikátory jednotlivým řetězcům tak, že nejsou závislé na aktuální pozici

[5] Standartní jazyk pro dotazování XML dokumentu, pomocí kterého lze vybírat jednotlivé elementy a pracovat s jejich hodnotami a atributy.

[6] Jazyk pro popis struktury XML případně SGML dokumentu. Struktura třídy nebo typu dokumentu je v DTD popsána pomocí popisu jednotlivých značek (nebo též elementů) a atributů.

ve zdrojovém souboru, jako je tomu u LZ77 viz 3.1.3. Díky tomu je algoritmus LZ77 nevhodný pro tento typ komprese dat. Hlavní kompresním algoritmem sémantického kompresoru „nahuff“ je neadaptivní Huffmanovo kódování. Pro zvýšení efektivity komprese se používají rozdílné tabulky četnosti znaků a to zvlášť pro atributy a zvlášť pro elementy. Tímto se bere ohled na sémantickou příbuznost dat ve stejných strukturách.

XGrind podporuje dotazování v komprimovaném souboru pomocí jazyka XML-QL[7]. Tyto dotazy jsou poté rozlišeny podle typu dotazu:

- Dotaz na přesnou shodu.
- Dotaz na shodu prefixu.
- Dotaz na částečnou shodu.
- Dotaz na rozsah.

Dotaz na **přesnou shodu** hledá značku nebo atribut přesně se shodující s hledaným výrazem. Příklad takového dotazu, který vyhledává všechny studenty, jejichž rollno = 123456789, se nachází na Výpisu číslo 4 . Dotaz na **shodu prefixu** hledá shodu prefixu značky s hledaným výrazem. Oba typy dotazů díky použití bezkontextové komprese komprimují cestu(od kořene k elementu) a predikát(hledaný výraz) dotazu stejným způsobem, jakým byla komprimována data. Soubor se tak nemusí při vyhodnocování těchto dvou typů dotazů dekomprimovat. XGrind využívá pro vyhodnocení dotazu bajtové zarovnání, což znamená, že porovnávání znaků dotazu a cílového souboru probíhá na úrovni bytů('A'=='B'), místo bitů(1==0). Tato metoda je rychlejší než bitové zarovnání, není však stejně efektivní.

```
CONSTRUCT <student rollno=$r> {
   WHERE
      <student rollno=123456789>
      <name>$n</name>
      <year>$y</year>
      <dept name=$d>
      </student> IN "student.xml",
   CONSTRUCT <name>$n</name>
}</student>
```

Výpis 4: Příklad dotazu na přesnou shodu.

U dotazů na **částečnou shodu** nebo **rozsah** se komprimuje pouze cesta dotazu. Pro porovnávání predikátu je třeba při shodě cesty dotazu data dekomprimovat. Příklad dotazu na rozsah lze nalézt na Výpisu číslo 5.

[7]XML Query Language je jazyk pro dotazování nad XML dokumenty. Stejně jako SQL používá SELECT-WHERE konstrukci.

```
CONSTRUCT <student rollno=$r> {
   WHERE
      <student rollno=$r>
      <name>$n</name>
      <year>$y</year>
      <dept name=$d>
      </student> IN "student.xml",
      $y >= 1998 and $y <= 2000
   CONSTRUCT <name>$n</name>
}</student>
```

Výpis 5: Příklad dotazu na rozsah.

Příklad 3.8. *Uvažujme následující XML dokument jako vstupní soubor:*

```
<!- student.xml -->
<STUDENT rollno = "604100418">
<NAME>Pankaj Tolani</NAME>
<YEAR>2000</YEAR>
<PROG>Master of Engineering</PROG>
<DEPT name = "Computer Science">
</STUDENT>
```

Poté bude DTD vypadat následovně:

```
<!- DTD for the Student database -->
<!ELEMENT STUDENT (NAME, YEAR, PROG, DEPT)>
<!ATTLIST STUDENT rollno CDATA #REQUIRED>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT YEAR (#PCDATA)>
<!ELEMENT PROG (#PCDATA)>
<!ELEMENT DEPT EMPTY>
<!ATTLIST DEPT name (Computer_Science| Electrical_Engineering| Physics |
    Chemistry)>
```

Značky a atributy budou zakódovány takto:

STUDENT = T0, NAME = T1, YEAR = T2, PROG = T3, DEPT = T4,

rollno = A0, name = A1, koncová značka = /

Data výčtových typů jsou označena jako enum(s), kdy s je vstupní řetězec. Ostatní data jsou označena jako nahuff(s). Tyto názvy určují, který kompresor bude použit pro kompresi těchto dat.

Tabulka 2: Sémantické kompresory XPress

kód	popis kompresoru
u8	kódování celých čísel C, kdy $C < 2^7$
u16	kódování celých čísel C, kdy $2^7 + 1 < C < 2^{15}$
u32	kódování celých čísel C, kdy $2^{15} + 1 < C < 2^{31}$
f32	kódování čísel s desetinnou čárkou
dict8	kódování výčtových dat
huff	Huffmanovo kódování textových dat

Výstupní soubor poté bude vypadat následovně:

T0 A0 nahuff(604100418)
T1 nahuff(Pankaj Tolani) /
T2 nahuff(2000) /
T3 nahuff(Master of Engineering) /
T4 A1 enum(Computer_Science) /
/

3.3.1.2 XPress XPress je dalším kompresorem zohledňujícím strukturu XML podporujícím dotazování, který byl vyvinut společností Microsoft. Jeho principy vychází z metody XGrind a dalo by se říct, že je jeho efektivnější verzí s lepšími principy dotazování. [12]

XPress využívá automatického odvození datových typů. Díky tomu je komprese efektivnější, než kdyby byl celý soubor zakódován pouze pomocí Huffmanova kódování (viz kapitola 3.1.2. Narozdíl od kompresoru XMill (více v kapitole 3.3.2.1), který také efektivně kóduje specifické datové typy, je XGrind samostatnější, jelikož datové typy detekuje automaticky bez nutného zásahu uživatele. Podporuje celkem 6 sémantických kompresů[8], z nichž 4 jsou použity pro kódování čísel a zbylé 2 pro kódování textu. Tyto kompresory jsou popsány v Tabulce číslo 2.

Komprese dat probíhá stejně jako u XGrind bezkontextově, což znamená, že kódování probíhá bez závislosti dat na pozici v souboru. Při kompresi je oddělena struktura od dat a pro její kompresi je využito reverzního aritmetického kódování, které je verzí aritmetického kódování, které je popsáno v kapitole 3.1.5. To přiřazuje každé cestě od kořenového elementu po právě kódovaný element interval v rozmezí 0 až 1. Tyto intervaly jsou pak rozděleny na menší podintervaly a ty jsou přiřazeny jednotlivým prvkům. Podle celkového počtu výskytů prvku se pak určuje velikost těchto intervalů. Na příkladu číslo 3.9 je tento postup vysvětlen.

Příklad 3.9. *Uvažujme vstupní soubor uvedený na Výpisu č. 6.*

```
<?xml version="1.0" encoding="utf-8" ?>.
<book>
```

[8] Algoritmus specializující se na kompresi sémanticky podobných dat, jako je třeba PSČ či čísla menší než 255.

```
   <author> author1 </author>
   <title> title1 </title>
   <section>
      <subsection> first subsection
         <subtitle> title3 </subtitle>
      </subsection>
   </section>
   <section>
      <subsection> subsection2 </subsection>
   </section>
   <section>
      <subsection> different subsection </subsection>
   </section>
</book>
```

Výpis 6: XML dokument

Pro dané elementy spočteme četnost výskytů a vytvoříme pro ně podintervaly pomocí principu aritmetického kódování (popsáno v kapitola 3.1.5). Četnosti výskytů jednotlivých elementů a jejich intervaly budou následující:

Element	*Četnost*	*Interval*
book	*1*	*<0.0 , 0.1)*
author	*1*	*<0.1 , 0.2)*
title	*1*	*<0.2 , 0.3)*
section	*3*	*<0.3 , 0.6)*
subsection	*3*	*<0.6 , 0.9)*
subtitle	*1*	*<0.9 , 1.0)*

Při dotazování nad souborem pro element „subsection“ tedy bude výsledný interval vypadat takto:

Element	*Cesta*	*Interval*	*Subinterval*
book	*book*	*<0.0 , 0.1)*	*<0.0 , 0.1)*
section	*book.section*	*<0.3 , 0.6)*	*<0.3 , 0.33)*
subsection	*book.section.subsection*	*<0.6 , 0.9)*	*<0.69 , 0.699)*

Při vyhodnocování dotazu budeme vyhodnocovat pouze data spadající do intervalu <0.69 , 0.699).

XPress využívá XPath dotazů. Při dotazování nad souborem se cesta dotazu vyhodnocuje pomocí intervalů. Cesta dotazu je zakódována na určitý interval stejným způsobem, jako byl zakódován dotazovaný soubor. Při vyhodnocení dotazu se počítá pouze s prvky, kde interval

cesty dotazu spadá do intervalu cesty právě těchto prvků. Tento způsob je efektivnější než XGrind, který vyhodnocuje všechny prvky, ať už interval cesty dotazu spadá do intervalu cesty prvků, či nikoliv.

3.3.1.3 XQzip XQzip je dalším z rodiny XML kompresních algoritmů s podporou dotazování. Jedná se o ještě lepší [4] variantu již zmíněných dvou kompresních metod. Při vývoji se vycházelo z nedostatků XPress a XGrind. XGrind při vyhodnocení dotazu musí projít celý XML dokument a pokud se shodují cesty prvku a dotazu, dotaz se vyhodnotí. XPress při vyhodnocování dotazů bere ohled pouze na podmnožinu intervalů odpovídající cestě dotazu díky aritmetickému kódování. Nicméně i tato podmnožina může být pořád obsáhlá, a to především u velkých XML dokumentů.

XQzip proto pracuje se strukturou SIT (Structure Index Tree). Pro čtení XML dokumentu je použit SAX parser (viz kapitola 2.4.2), který při prvním čtení souboru zároveň vytváří SIT. Příklad SIT lze vidět na Obrázku č. 5. XQzip modeluje strukturu XML dokumentu jako strom. Tento strom obsahuje pouze kořenový uzel a uzly pro elementy. Tyto elementární uzly představují obojí atributy i elementy, pro rozlišení se poté dává "@"na začátek jména atributu. Každému jedinečnému elementu či atributu je poté přiřazen vlastní index z hashovací tabulky, do které jsou tyto prvky uloženy. XML data jsou poté oddělena od struktury a jsou komprimována zvlášť pomocí metody GZip[9] (viz kapitola 3.2.1) v blocích, které jsou přístupné právě pomocí již zmíněné hashovací tabulky.

Díky tomu je vyhodnocování dotazů efektivnější, jelikož se prohledává pouze optimalizovaná struktura SIT a k prvkům tak lze přistupovat přímo pomocí hashovací tabulky. Navíc se při dotazování využívá vyrovnávací paměť, která urychluje provádění stejných či podobných úkonů. V poslední řadě pak XQzip podporuje větší množství XPath dotazů, než je tomu u jeho předchůzců XPress a XGrind (který podporuje pouze XML-QL), čímž nabízí větší možnosti dotazování v komprimovaných souborech.

3.3.2 XML komprese bez podpory dotazování

Kompresní algoritmy spadající do této kategorie využívají sémantické informace získané ze struktury XML. V případě, že je potřeba dotazování, se narozdíl od algoritmů podporujících dotazování musí celý soubor nejdříve dekomprimovat a zpracovat. V případě, že bychom chtěli provést nějakou aktualizaci, následně musíme soubor opět zkomprimovat. Cílem těchto kompresorů je co nejlépe připravit XML dokument k následné kompresi některou ze standartních metod pro kompresi textu (viz kapitola 3.2).

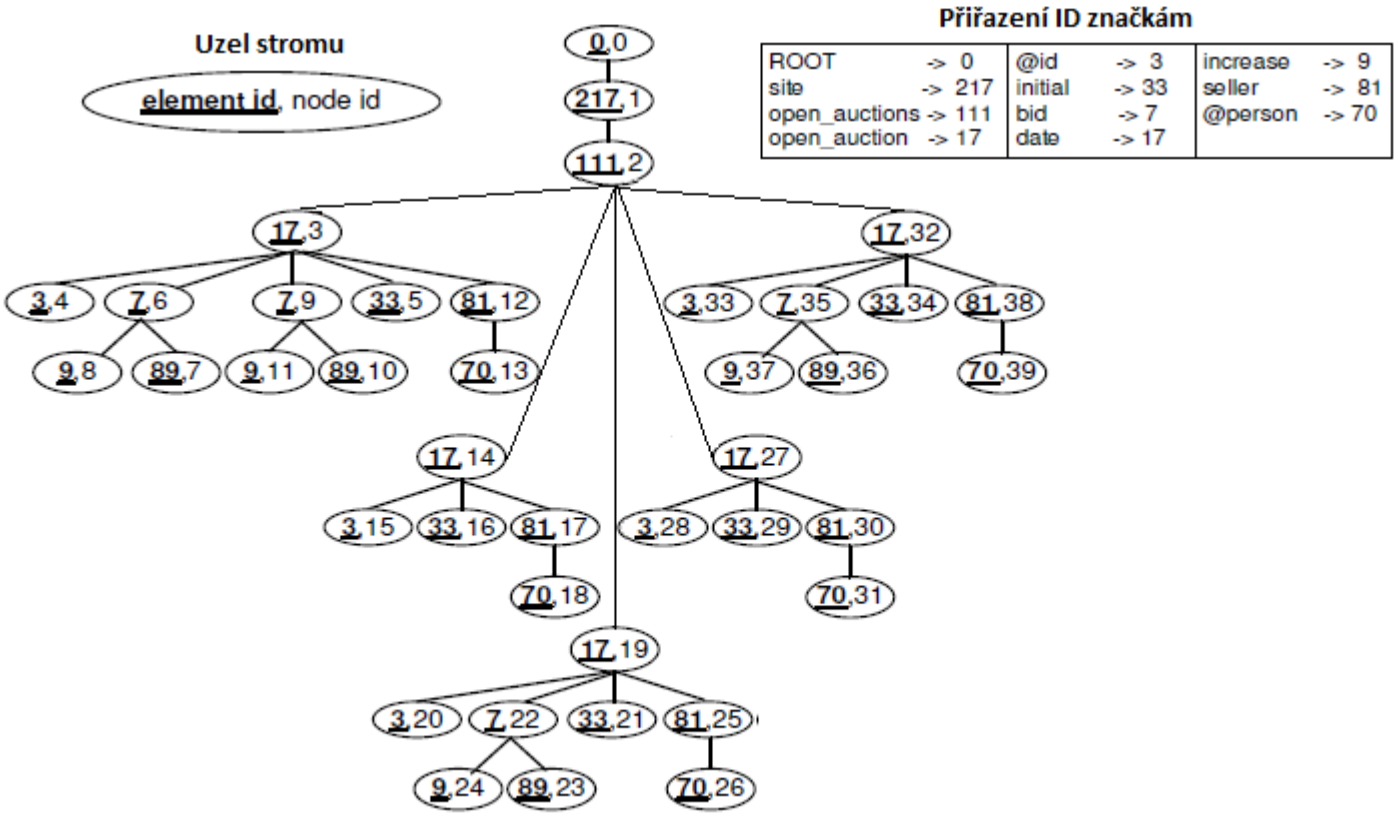
3.3.2.1 XMill XMill je metoda komprese XML dat bez podpory dotazování. U větších souborů dosahuje většinou lepších výsledků, v porovnání s GZip (viz 3.2.1) dosahuje až dvakrát lepších výsledků [11]. Samotný nástroj XMill pracuje z příkazové řádky a prochází soubory

[9] Odtud název XQzip

Rozdělení souboru podle značek

```
1. <site>
2. <open_auctions>
3.  <open_auction id="open1">
4.   <initial>$12.00</initial>
5.   <bid>
6.    <date>12/02/2000</date>
7.    <increase>$2.00</increase>
8.   </bid>
9.   <bid>
10.   <date>12/03/2000</date>
11.    <increase>$1.50</increase>
12.   </bid>
13.   <seller person="person71"/>
14.  </open_auction>
15.  <open_auction id="open2">
16.   <initial>$500.00</initial>
17.   <seller person="person8"/>
18.  </open_auction>
19.  <open_auction id="open3">
20.   <initial>$1.50</initial>
21.   <bid>
22.    <date>11/29/2002</date>
23.    <increase>$0.50</increase>
24.   </bid>
25.   <seller person="person15"/>
26.  </open_auction>
27.  <open_auction id="open4">
28.   <initial>$100.00</initial>
29.   <seller person="person11"/>
30.  </open_auction>
31.  <open_auction id="open5">
32.   <initial>$8.50</initial>
33.   <bid>
34.   <date>08/20/2002</date>
35.    <increase>$5.00</increase>
36.   </bid>
37.   <seller person="person7"/>
38.  </open_auction>
39. </open_auctions>
40. </site>
```

Struktura stromu



Obrázek 5: Vytvoření struktury SIT metody XQzip. Obrázek převzat z [4]

Tabulka 3: Sémantické kompresory XMill

kód	popis kompresoru
t	výchozí textový kompresor
u	kompresor pro kladná čísla
i	kompresor pro jakákoliv čísla
u8	kompresor pro kladná čísla < 256
di	delta kompresor pro jakákoliv čísla
rl	RLE kompresor (viz 3.1.1)
"..."	konstantní kompresor

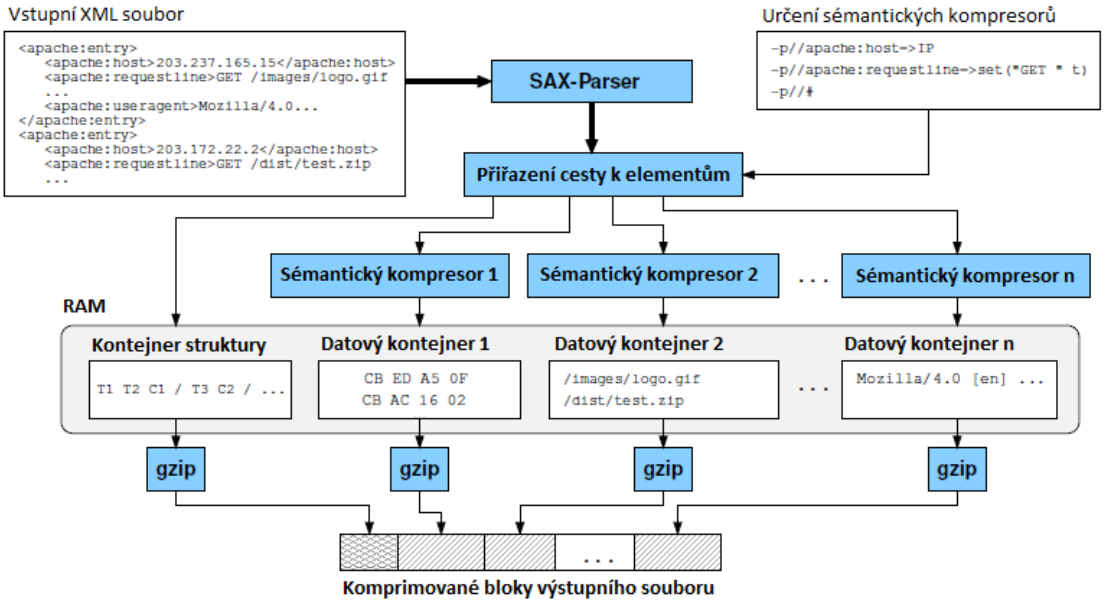
jednotlivě, nezvládá jich více najednou. Soubory s koncovkou *.xml ukládá po komprimaci do formátu *.xmi. XMill komprimuje strukturu a data odděleně. XMill pracuje na principu shlukování a komprimování textových položek založeném na jejich sémantice. Pokud máme například element <osoba>, která má podelementy <jméno> a <věk>, tyto elementy a jejich obsahy mohou být přeskupeny tak, že všechna jména a věk budou seskupeny spolu do datových kontejnerů. Datový kontejner si lze představit jako strukturu pro ukládání dat elementů a atributů s možností provádění různých operací nad těmito daty, jako je aplikování sémantických kompresorů apod.

Řetězec označující věk lze například nahradit jeho binární reprezentací. Komplexněji se poté dá například IP adresa nahradit pomocí 4 bytové reprezentace. Tato vlastnost může vést k lepšímu kompresnímu poměru, jelikož každá skupina bude obsahovat textové položky s velkou podobností. XMill tak aplikuje různé sémantické kompresory na různé datové kontejnery s ohledem na jejich sémantiku, tudíž každý takový datový kontejner je komprimován zvlášť. XMill podporuje také uživatelsky vytvořené sémantické kompresory. V tabulce číslo 3 jsou popsány již existující XMill sémantické kompresory.

Uživatel může specifikovat, jakým způsobem před-komprimovat specifickou textovou položku, tedy jaké využít sémantické kompresory. V každém případě je po provedení této komprese vždy aplikována kompresní metoda GZip, která je popsána v kapitole 3.2.1. Jelikož počet a velikost úložných kontejnerů roste s velikostí XML souboru, je implementován mechanizmus paměťového okna. Jakmile celková velikost těchto kontejnerů dosáhne určité uživatelsky specifikované meze, obsah kontejnerů je komprimován a uložen do výstupního souboru a poté jsou tyto kontejnery vyprázdněny a zaplňovány odznova. Nevýhodou XMill je však použitelnost, jelikož u souborů menších než 20 KB nedosahuje dobrého kompresního poměru.

Na obrázku 6 je názorně ukázáno, jakým způsobem a v jakém pořadí probíhá komprese.

Komprese Při kompresi se nejprve zjistí, jestli uživatel nezadal nějaký specifický kompresor, například, aby se čísla představující obsah elementů s určitým názvem zapisovala v binární podobě a podobně. Pomocí SAX parseru (více v kapitole 2.4.2) se soubor rozdělí do kontejnerů podle jmen elementů. Hodnoty v elementu <jmeno> tak budou seskupeny ve stejném kontej-



Obrázek 6: Vysvětlení rozdělení úloh při kompresi pomocí XMill. Obrázek převzat z [11]

neru, <prijmeni> se pak budou nacházet v jiném. Na hodnoty ukládané do těchto kontejnerů jsou pak aplikovány sémantické kompresory, které jsou specifické pro typ dat ukládaných do těchto kontejnerů. Struktura XML dokumentu je také uložena do zvláštního kontejneru. Na všechny tyto kontejnery jsou poté aplikovány kompresní algoritmy LZ77 a Hufmanovo kódování a to postupně, začínaje kontejnerem obsahujícím strukturu. Ve výstupním souboru jsou tedy data zakódována za sebou podle jejich typu v blocích. Celý vstupní soubor je pro potřebu dekomprese zakódován tak, že nahrazuje právě čtený element/atribut/data značkou, která odkazuje na kontejner, ve kterém se právě čtená data nachází. Všechny tyto značky jsou zakódovány pomocí 1 B, 2 B nebo 4 B binární hodnotou následovně:

- **Elementy a atributy** jsou zakódovány kladným číslem.
- **Data** jsou zakódována záporným číslem.
- **Koncové značky** jsou zakódovány nulou.

Pro elementy a atributy se používají kladná čísla, pro obsah se používají záporná čísla a koncové značky jsou zakódovány nulou. Princip komprese je vysvětlen na příkladu číslo 3.10.

Dekomprese Při dekompresi se nejprve na soubor aplikuje dekomprese pomocí GZip, tedy Huffmanovo kódování a poté LZ77. Dekódovaný obsah je poté rozdělen do datových kontejnerů podle typu dat v takovém pořadí, v jakém byly zakódovány. Tyto kontejnery jsou poté dekomprimovány pomocí stejných sémantických kompresorů, které byly použity při kompresi. Při čtení zakódované sekvence jsou poté odkazy nahrazeny hodnotami obsaženými v těchto datových kontejnerech.

Příklad 3.10. *Na ukázku fungování tohoto algoritmu berme následující text jako vstupní XML soubor:*

```
<?xml version="1.0" encoding="utf-8" ?>.
<Book> <Title lang="English"> Views </Title>
<Author> Miller </Author>
<Author> Tai </Author>
</Book>
```

Značky budou nahrazeny těmito odkazy na datové kontejnery:
Book = T1, Title = T2, @lang = T3, Author = T4, Koncová značka pomocí /
Data budou rozdělena do kontejnerů následovně:

Kontejner č.1	*Kontejner č.2*	*Kontejner č.3*	*Kontejner č.4*
Book	*Title* *Views*	*lang* *English*	*Author* *Miller* *Tai*

Poté celý soubor bude zakódován touto sekvencí: T1 T2 T3 C3 / C2 / T4 C4 / T4 C4 / /
Toto však není kompletní výsledek, ale pouze výstup souboru před komprimační fází, jelikož tyto odkazy budou přepsány do binární podoby, stejně tak poté budou komprimovány pomocí GZip.

4 Testovací aplikace

Pro účely testování vybraných kompresních algoritmů byly algoritmy naimplementovány do jedné aplikace nazvané XCom v programovacím jazyce C++, který poskytuje lepší možnosti fyzické optimalizace kódu oproti jazykům, jako jsou Java nebo C# [8]. XCom je tedy aplikace, která slouží pro kompresi a dekompresi datových souborů pomocí různých, námi implementovaných, základních kompresních principů a jejich kombinací.

Vstupem této aplikace je soubor, který chceme zkomprimovat a výstupem je zkomprimovaný soubor v jiném datovém formátu, než byl jeho originál. Tato aplikace navíc vytváří hlášení o kompresi, kdy se do samostatného souboru uloží informace o kompresním poměru, velikostech originálního a nového souboru, jaký kompresní algoritmus byl použit a jak dlouho operace komprese nebo dekomprese trvala. Díky tomuto hlášení lze jednoduše jednotlivé metody porovnat. V aplikaci jsou naimplementovány všechny algoritmy, které jsou vyjmenovány v 5.1.

Aplikace se dá spustit z příkazové řádky pomocí zadání parametrů, či se dá otevřít samotně a lze s ní pracovat pomocí jednoduchého uživatelského rozhraní. Popis pro práci s programem lze nalézt v příloze A.

4.1 Návrh

Aplikace je rozdělena na několik hlavních částí a to na třídy pro práci se souborem, hlavní spouštěcí třídu a rámec obsahující algoritmy. Aplikace byla naimplementována s ohledem na možné budoucí rozšíření o další kompresní algoritmy.

Popis tříd:

4.1.1 Třídy pro práci se souborem:

Tyto třídy jsou určeny pro čtení či zápis do souboru. Snažíme se tak rozdělit architekturu aplikace a umožnit práci se soubory bez nutnosti implementace kódu pro práci se souborem všude tam, kde je to potřeba. Místo toho tyto třídy poskytují jednoduché rozhraní, pomocí kterého lze se souborem pracovat.

`InputBitStream` je třída sloužící k otevírání a čtení souborů. Umí číst soubor jak po bytech, tak po bitech. Tato třída se tedy stará o veškerou práci při čtení dat ze souboru a zbytku aplikace poskytuje metody, pomocí kterých mohou tato data získávat. Hlavní metody této třídy pro práci se souborem jsou:

- `Next`() - Tato metoda slouží ke čtení 1 bytu dat ze souboru. Tato vrací znak (datový typ `char`) představující přečtený znak.
- `readBitasNumber`() - Tato metoda přečte 1 byte ze vstupního souboru. Tento znak je poté převeden na číselnou hodnotu, kterou byl vyjádřen v bitové podobě. Metoda vrací číslo v podobě datového typu `unsigned long`.

`OutputBitStream` je třída sloužící k otevírání a zápisu do souborů. Má naimplementovány metody pro zápis do souboru jak po bytech, tak po bitech. Při zápisu po bitech čeká, než se načte 8 bitů, které jsou poté zapsány jako 1 byte. Pokud je pro zápis 1 znaku použito méně, než 8 bitů, jsou tyto bity doplněny nulami. Hlavní metody této třídy pro práci se souborem jsou:

- `WriteByte`(< `t` > `C`) - Tato metoda slouží k zápisu 1 bytu dat do souboru. Jako vstupní parametr bere jakékoliv číslo či znak, jelikož má tato metoda několik přetížení. Pod `<t>` si tedy lze představit datové typy char, int, double apod.
- `WriteBits`(`vector` < `bool` > `bits`) - Tato metoda zapisuje do souboru bity. Vstupem této metody je vector s boolean hodnotami. Pokud je boolean hodnota true, zapíše do souboru 1. Pokud false, zapíše 0. Bity jsou zapisovány po 8 bitech - 1 bytu.

4.1.2 Hlavní spouštěcí třída:

Do této kategorie spadá třída obsahující metodu `Main()` a byla zde přidána i třída, pomocí které tato metoda interaguje s uživatelem.

`Xcom` je zároveň název aplikace a zároveň název třídy implementující metodu `Main()`. V této třídě jsou implementovány metody na výpis hlášení o kompresi, které nám pomáhá jednotlivé algoritmy srovnávat. V samotné metodě `Main()` jsou nejprve nainicializány všechny potřebné prvky aplikace, jako jsou například používané algoritmy, dále se zde nachází už pouze cyklus, který se stará o spouštění algoritmů a zároveň interaguje s uživatelem. Tento cyklus byl navržen tak, aby nebylo nutné pro každou možnost psát zvláštní část kódu a to především tam, kde by se kód opakoval a lišil pouze v detailech. Tohoto dosahujeme především proto, že všechny algoritmy jsou uloženy v seznamu a lze je spustit pomocí stejného principu. Tento princip je názorně ukázán na Algoritmu číslo 7.

Na řádku číslo 1 je vidět vytváření seznamu kompresních algoritmů, které jsou do něj přiřazeny na řádcích 2-7. Na řádku 8 je poté zavolána metoda `Compress`() vybraného algoritmu.

Algoritmus 7: Princip spouštění algoritmů.

```
1 CompressionAlgorithm * CA[počet algoritmů];
2 CA[0] = newRLE();
3 CA[1] = newMTF();
4 CA[2] = newHuffman();
5 CA[3] = newArithmetic();
6 CA[4] = newLZ77();
7 CA[5] = newXMill();
8 CA[Vstup uživatele − 1] → Compress(InputBitStream, OutputBitStream);
```

`Print` je třída zajišťující výpis hlášek pro uživatele. Jelikož je při výpisu odkazováno na tuto třídu, stačí text v případě potřeby změnit na jednom místě.

4.1.3 Rámec kompresních algoritmů:

Pro všechny algoritmy bylo vytvořeno společné rozhraní.

`CompressionAlgorithm` je abstraktní třída, ze které dědí všechny třídy obsahující algoritmy. S algoritmy se poté pracuje pomocí tohoto rozhraní a to především pomocí metod `Compress(InputBitStream& is, OutputBitStream& os)`
a `Decompress(InputBitStream& is, OutputBitStream& os)`, které musí všechny třídy z ní dědící implementovat. Vstupem pro tyto metody jsou instance tříd `InputBitStream`
a `OutputBitStream`. Výstupem je poté cesta k výstupnímu souboru. Takto se totiž dají algoritmy mezi sebou řetězit. V aplikaci je poté v seznamu uložena instance každého z algoritmů, jehož metody jsou spouštěny přes rozhraní této abstraktní třídy. Třídní diagram pro tyto třídy a rozhraní, které implementují, naleznete na obrázku číslo 7

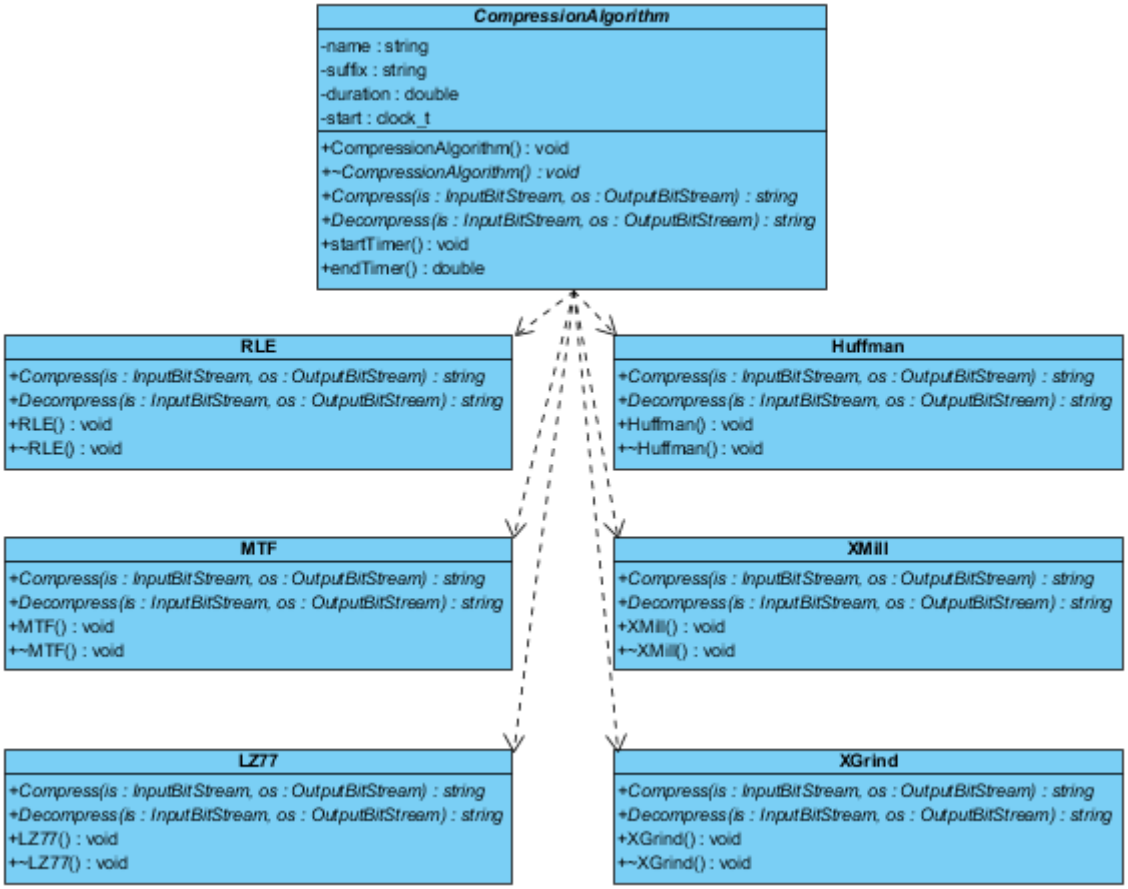
`RLE` je třída, která implementuje algoritmus Run-Lenght encoding popsaný v kapitole 3.1.1. Jako zvláštní znak pro tento algoritmus byl zvolen znak '@' a zakódovaná sekvence znaků je tak reprezentována pomocí "@5a" kdy "@" je označení komprimovaných dat, "a" je komprimovaný znak a "5" je počet opakování znaku zapsaných v binární podobě. V této verzi implementace nemůže být počet opakování větší, než 256, jelikož počet opakování je zakódován pomocí 8 bitů.

`MTF` je třída implementující algoritmus Move To Front transform popsaný v kapitole 3.1.4. Vstupní abecedou pro mapu jsou všechny znak, jejichž binární hodnota je vyjádřena čísly 0 - 255. Jako výstup je každé číslo uloženo v binární podobě.

`Huffman` je třída se statickou verzí algoritmu Huffmanovo kódování, který je popsán v kapitole 3.1.2. V komprimovaném souboru se nejprve nachází reprezentace huffmanova stromu, která je mezerou oddělena od bitové reprezentace komprimovaného souboru.

`LZ77` je třída s implementací algoritmu Lempel-Ziv 1977. Tento algoritmus byl popsán v kapitole 3.1.3.

Obrázek 7: Třídní diagram.



5 Testování komprese XML dokumentů

Tato část práce je zaměřena na testování komprese XML dokumentů, popisuje jaký stroj byl k tomuto účelu využit, jaké XML kolekce byly použity a také výsledky komprese.

5.1 Parametry testování

Testovací algoritmy

Pro testování komprese XML souborů byly základní kompresní algoritmy a kompresní nástroje s ohledem na XML strukturu. Ze základních algoritmů pro kompresi dat to jsou:

- **Move To Front transform** – typ transformace, který mění znaky vstupního soubory za číselné hodnoty (viz kapitola č. 3.1.4).
- **Huffmanovo kódování** – kóduje nejčastěji se vyskytující znaky pomocí co nejmenší bitové reprezentace (viz kapitola č. 3.1.2).
- **Lempel-Ziv 1977** – kóduje opakující se řetězce do posloupnosti <vzdálenost od předchozího výskytu, délka sekvence, následující znak> (viz kapitola č. 3.1.3).
- **Run length encoding** – kóduje za sebou jdoucí znaky do poslouponosti <počet opakování, znak> (viz kapitola č. 3.1.1).

Z algoritmů specializovaných na kompresi XML dokumentů byly vybrány:

- **XGrind** – komprese XML s ohledem na strukturu s možností dotazování (viz kapitola č. 3.3.1.1).
- **XMill** – komprese XML s ohledem na strukturu bez možnosti dotazování (viz kapitola č. 3.3.2.1).

Obě tyto kompresní metody podporují sémantické kompresory. V rámci testů a porovnávání s běžnými textovými metodami jsme při kompresi uvažovali data elementů a atributů jako textové položky.

Testovací kolekce

Jako testovací kolekce byly vybrány již existující volně dostupné XML dokumenty a to:

1. **XMark** [16] – syntetická databáze aukcí s velkou úrovní zanoření elementů. Lze vygenerovat různou velikost souboru.[10] Vygenerováno bylo několik souborů o různých velikostech. Detailně se této kolekci věnujeme v kapitole 5.2.4

[10] Projekt byl ukončen v březnu 2018.

2. **TreeBank** – zakódované anglické věty, silně nepravidelná a rekurzivní povaha dokumentu. Velikost souboru je 82,1 MB.[11]

3. **Lineitem** – databáze objednávek z internetového obchodu. Soubor má velikost 30,7 MB. [12]

4. **Mondail** – kolekce informací o jedntlivých státech a městech. Velikost souboru je 1,7 MB.[13]

Testovací stroj

Testy komprese byly prováděny na testovacím stroji s následujícími parametry:

Operační systém	Windows 10 Home 64bit
Procesor	Intel Core i5-2450p 4x @3,20GHz
Paměť RAM	8 GB DDR3
Úložné médium	SSD Kingston SV300S37A120G

5.2 Výsledky testování

Výsledky testování potvrdily, že kompresní metody, které berou ohled na strukturu XML dokumentu, mají lepší kompresní poměr, než běžné kompresní metody. Každý z testovaných souborů byl jednotlivými kompresními metodami a algoritmy komprimován i dekomprimován celkem deset krát. Jako naměřenou hodnotu jsme poté počítali s průměrem těchto deseti hodnot. Zaměřovali jsme se především na čas komprese, čas dekomprese a kompresní poměr.

Jak již bylo zmíněno v kapitole 3.1.4, Move To Front transformation neprovádí samo o sobě kompresi, pouze mění vstupní data. Zkusili jsme tedy otestovat, jestli při kombinaci tohoto algoritmu společně s RLE(viz kapitola 3.1.1) dosáhneme lepšího výsledku komprese, než při použití samotného algoritmu RLE.

Všechny běžné kompresní principy po provedení komprese a následně zpětné dekomprese vrátily stejný soubor. Naše implementace metod XMill a XGrind (viz kapitoly č. 3.3.2.1 a 3.3.1.1) ignorují bílé znaky[14].

5.2.1 TreeBank

TreeBank je kolekce obsahující zakódované anglické věty. Dokument má nepravidelnou a rekurzivní povahu, přičemž elementy se mohou zanořovat až do hloubky 36. Nejlepší komprimační poměr pro tuto kolekci měl algoritmus XGrind a to 0.41, jelikož zkomprimovaný soubor má 33,77 MB. Časově však se svými 15,7 vteřinami dopadl jako druhý, jelikož nejrychleji soubor

[11] Ke stažení na `http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/data/treebank/treebank_e.xml`

[12] Ke stažení na `http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/data/tpc-h/lineitem.xml`

[13] Ke stažení na `http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/data/mondial/mondial-3.0.xml`

[14] Znak, který představuje prázdné místo. Například tabulátor, mezerník či nový řádek.

zkomprimovat algoritmus RLE, a to za 7,5 sekund. XMill, také xml kompresor, dosáhl kompresního poměru 0.56 a umístil se tak na druhém místě. Bohužel XMill byl však nejpomalejším kompresorem, jelikož mu komprimace trvala 32 sekund. Časy komprimace a úsporu místa jednotlivých algoritmů lze najít na Obrázku číslo 8. Kombinace RLE a MTF nepřinesla lepší výsledek. Naopak byl čas potřebný pro provedení operace dvakrát větší a kompresní poměr byl mírně horší.

5.2.2 Lineitem

Lineitem je kolekce obsahující informace o jednotlivých objednávkách z internetového obchodu. Soubor má pravidelnou strukturu a úroveň zanoření je maximálně 3. Na této kolekci lze pozorovat rozdíl mezi běžnými kompresními metodami a těmi určenými ke kompresi XML dat, jelikož oba tyto kompresory dosáhly úspory místa více než 73%, v případě XGrind dokonce necelých 74,7%. Co se času potřebného pro kompresi týče, i tady XML kompresory dosahovaly lepších výsledků. Nejrychli kompresi provedl algoritmus RLE, ten však nedosáhl takřka žádné komprese souboru. Při kombinaci RLE s MTF se však potvrdilo, že MTF může v určitých případech pomoci zlepšit kompresní poměr, jelikož úspora místa zde dosahovala hodnoty 8%. Kombinace těchto dvou algoritmů však trvala nejdéle. Výsledky komprese nad touto kolekcí lze nalézt na Obrázku číslo 9.

5.2.3 Mondial

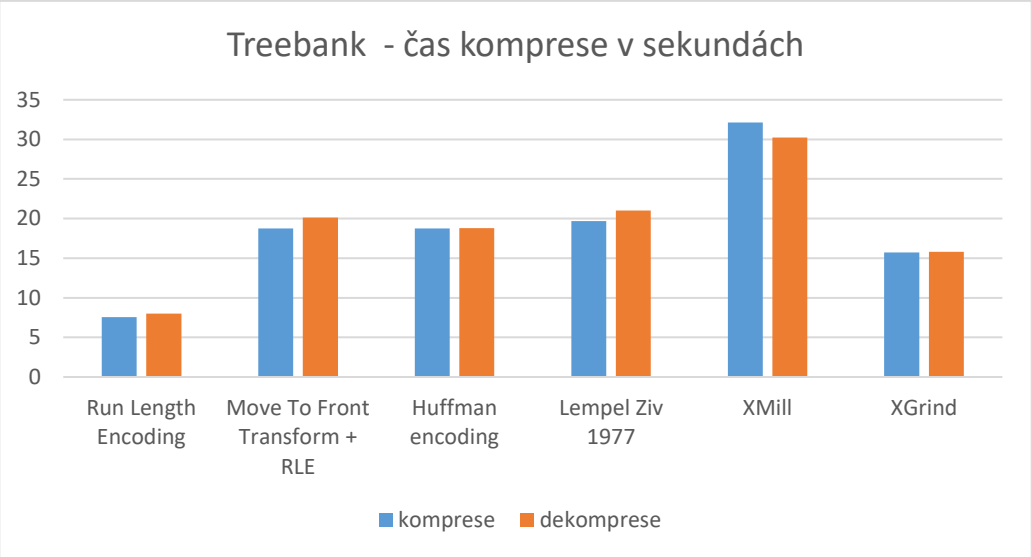
Modial je nejmenší kolekcí ze všech testovaných kolekcí. Tato kolekce obsahuje informace o jednotlivých státech a kontinentech a jejich populaci, státním zřízení apod. Většina dat je umístěna v atributech. Nejlepších výsledků dosáhl kompresor XMill, jelikož se svým kompresním poměrem 0,22 dosahuje téměř 78% úspory místa. Oproti ostatním metodám však potřebuje až čtyři krát více času. Vezmeme-li v potaz poměr rychlost/komprese, nejlépe dopadl opět XGrind, kterému 72,4% úspora místa trvala 0.3 vteřiny pro kompresi a 0.2 vteřiny pro dekompresi. Efektivní byl nad touto kolekcí i algoritmus LZ77, jehož úspora místa dosáhla hodnoty 64,3% a dekompresi zvládl za 0.1 vteřiny. Výsledky komprese kolekce Mondial jsou znázorněny na Obrázku číslo 10.

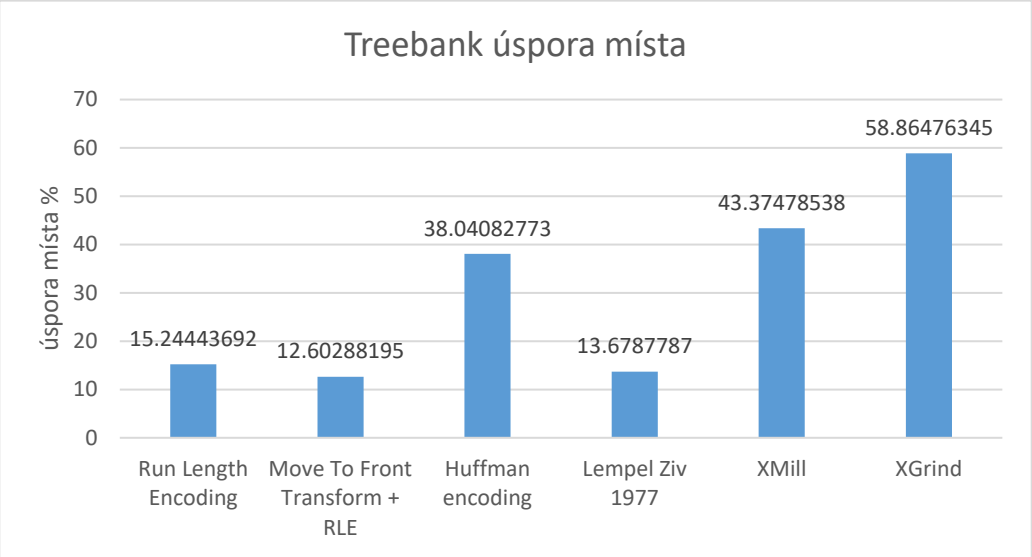
5.2.4 XMark

XMark je aplikace, pomocí které lze vytvořit XML kolekce o různých velikostech podle zvoleného faktoru. Kolekce obsahuje infomace o výrobcích a elektronických zprávách. V datech se mohou objevovat vnořené elementy. Pro účely testování jsme vytvořili několik kolekcí s následujícími vlastnostmi:

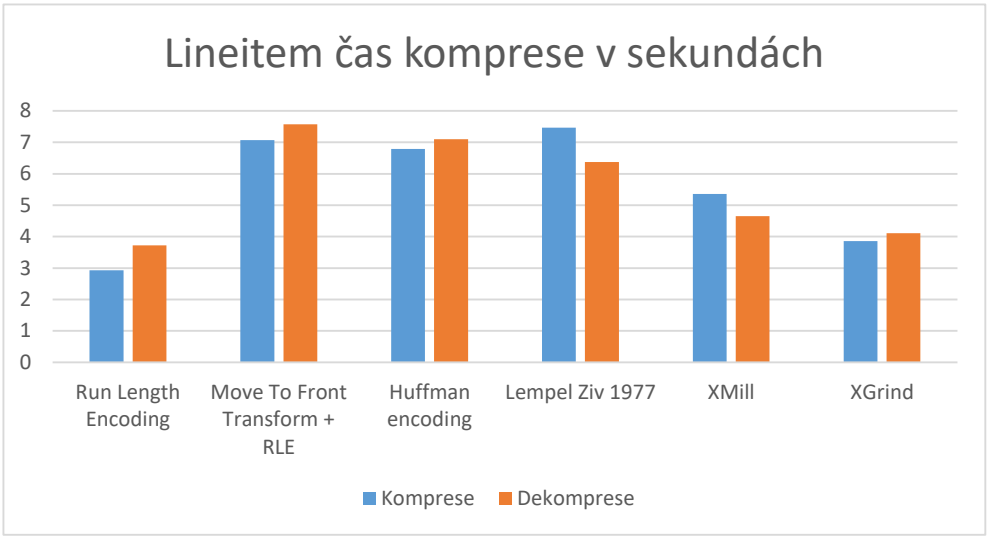
- faktor = 0.2, velikost = 22.8 MB
- faktor = 0.5, velikost = 56,2 MB
- faktor = 1, velikost = 113 MB

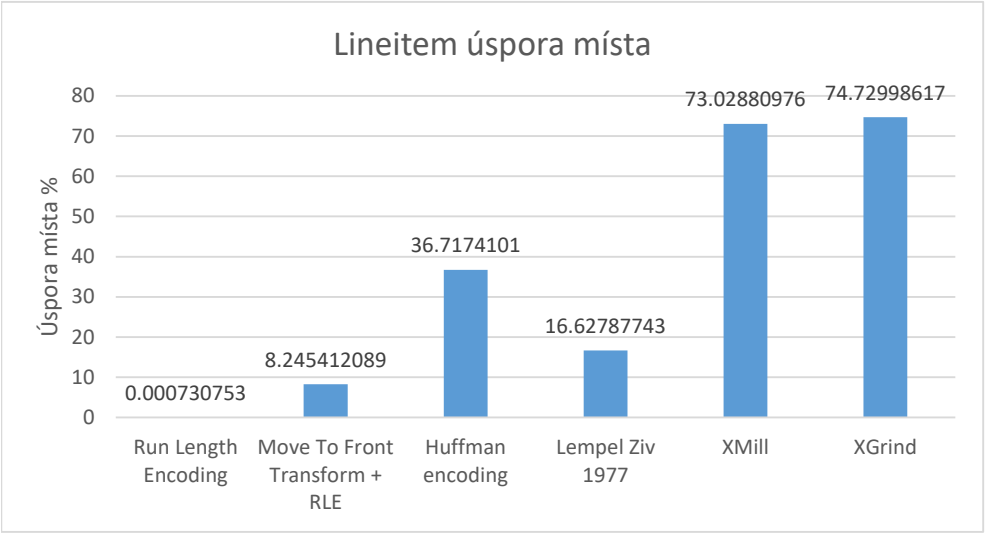
Obrázek 8: Výsledky komprese kolekce Treebank vyjádřené časem komprimace a dekomprimace v sekundách a úsporou místa v procentech pro jednotlivé algoritmy.



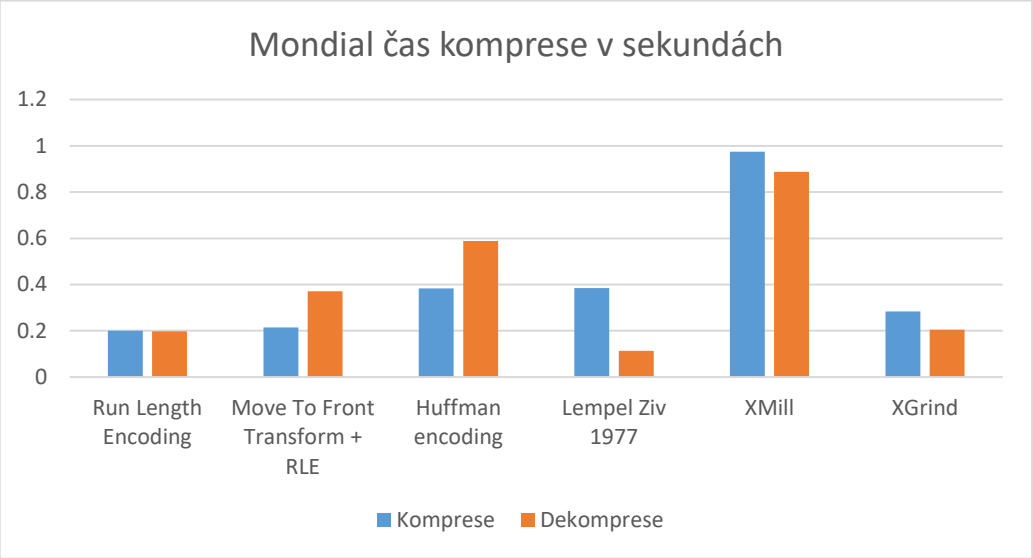


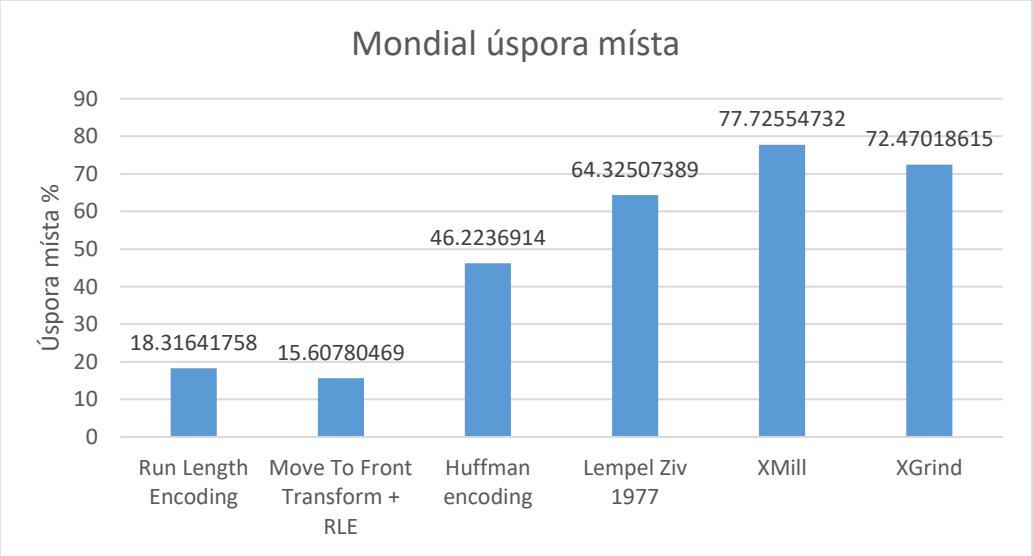
Obrázek 9: Výsledky komprese kolekce Lineitem vyjádřené časem komprimace a dekomprimace v sekundách a úsporou místa v procentech pro jednotlivé algoritmy.





Obrázek 10: Výsledky komprese kolekce Mondial vyjádřené časem komprimace a dekomprimace v sekundách a úsporou místa v procentech pro jednotlivé algoritmy.





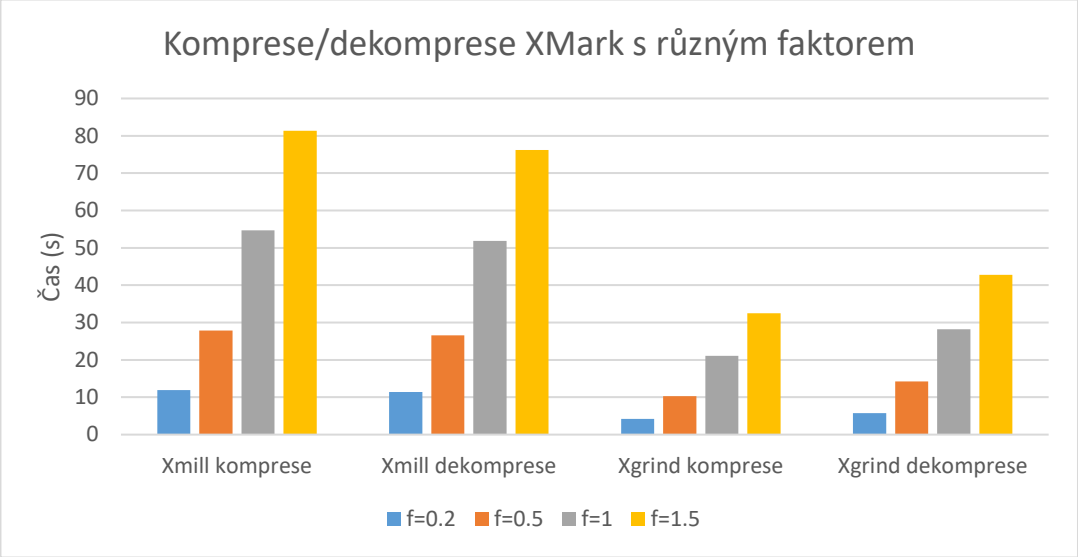
- faktor = 1.5, velikost = 170 MB

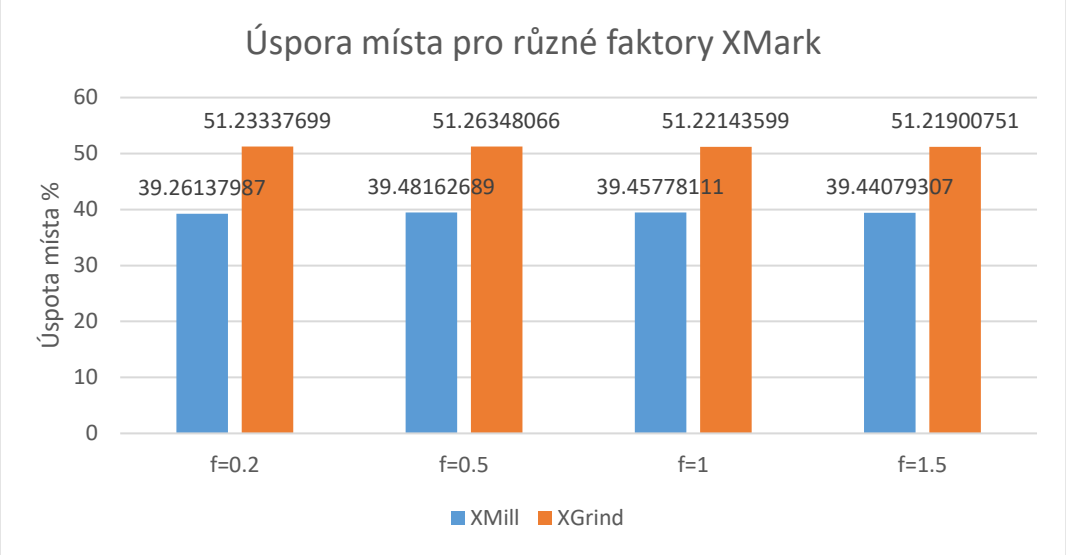
Nad touto kolekcí jsme testovali pouze algoritmy XMill a XGrind, jelikož se specializují na strukturu XML. Výrazně lepší výsledků dosáhl XGrind, který provedl kompresi i dekompresi více než dvakrát rychleji. Kompresní poměr má Xgrind také lepší, jelikož se komprimovaný soubor zmenšil o 51%, zatímco v případě XMill to bylo 39%. Čas potřebný pro kompresi a dekompresi lineárně rostl s velikostí souboru. Rozdíl v kompresním poměru je však pro různé velikosti kolekce zanedbatelný. Výsledky komprese kolekce XMark vygenerované s různými faktory jsou znázorněny na Obrázku číslo 11.

5.3 Kompresory XML

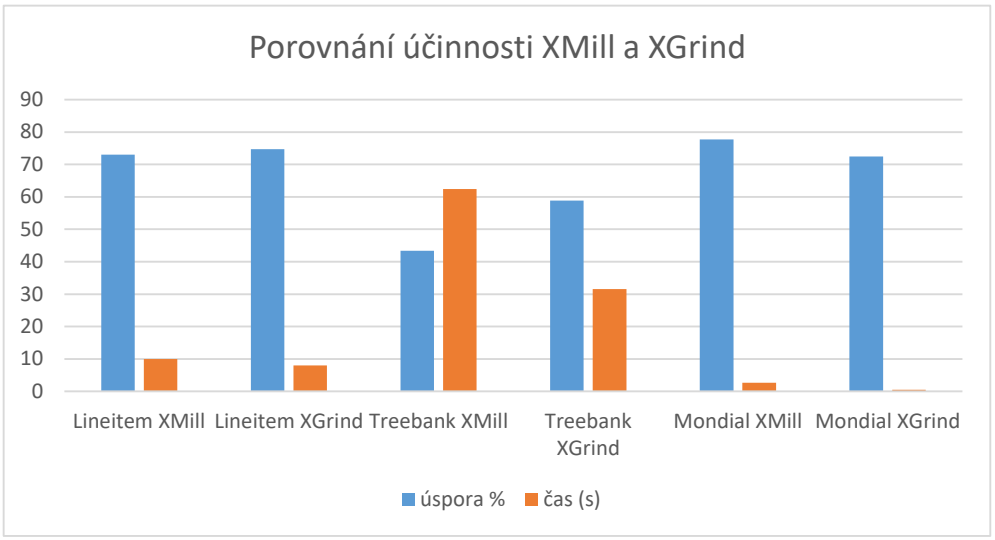
Z výsledků vyplývá, že kompresory, které berou ohled na strukturu XML dokumentů, dosahují lepších výsledků, než běžné kompresní algoritmy. Námi naimplementovaný kompresor XGrind dosahoval lepších výsledků, než naše implementace kompresoru XMill. XMill by sice mohl při použití sémantických kompresorů dosahovat lepších výsledků komprese, díky využití dvou kompresních principů však potřebuje více času ke kompresi. Srovnání výsledků komprese těchto dvou kompresorů lze nalézt na Obrázku číslo 12.

Obrázek 11: Výsledky komprese kolekce XMark vyjádřené časem komprimace a dekomprimace v sekundách a úsporou místa v procentech pro různé faktoriály.





Obrázek 12: Výsledky komprese a dekomprese kompresních metod XMill a XGrind.



6 Závěr

Cílem práce bylo nastudovat možnosti komprimace XML dat, vybrané algoritmy naimplementovat a porovnat je jak z pohledu komprimačního poměru, tak z pohledu času komprimace a dekomprimace. Práce začíná popisem značkovacího jazyku XML, který charakterizuje, popisuje jeho výhody ale i nedostatky a také se věnuje různým přístupům ke zpracovávání těchto dokumentů.

Popisu jednotlivých algoritmů a metod kombinujících jednotlivé algoritme je věnovaná celá, nejobsáhlejší kapitola. Je zde uvedena základní charakteristika každého z algoritmů. Dále jsou zde popsány principy komprese, dekomprese, které jsou názorně vysvětleny na příkladech doplněných pseudokódy. Jsou popsány jak základní algoritmy pro kompresi dat, tak algoritmy specializující se na kompresi XML dokumentů.

Vybrané algoritmy byly naimplementovány do testovací aplikace, které je také věnována samostatná kapitola. Tato aplikace využívá rozhraní pro lepší práci s algoritmy a pro její vývoj bylo využito jazyka C++, který poskytuje lepší možnosti fyzické optimalizace kódu oproti jiným programovacím jazykům.

Závěrem bylo provedeno testování vybraných algoritmů, a to vždy alespoň desetkrát pro každý pokus měření, abychom zvýšili přesnost měření. Výsledky měření jsou popsány v poslední kapitole. Z výsledků je zřejmé, že XML dokumenty se dají komprimovat jak pomocí běžných kompresních algoritmů a nástrojů užívaných pro kompresi textových souborů, tak i především pomocí kompresorů, které zohledňují strukturu XML dokumentu. Při našem měření jsme ve většině případů dosáhli pomocí těchto specializovaných kompresorů kompresního poměru alespoň $0,5$. Podle naměřených výsledků lze prohlásit, že kompresní metody zohledňující XML strukturu dosahují lepších výsledků komprese XML dokumentů, než běžné kompresní algoritmy.

Zvláštní pozornost si poté zaslouží Huffmanovo kódování, které bylo v každém případě komprese účinné a vždy snížilo vstupní soubor alespoň o 33%. Díky tomu lze říct, že je to velice účinný kompresní algoritmus pro širokou škálu vstupních souborů. Dále bylo dokázáno, že některé algoritmy jsou v určitých případech v kombinaci s jinými algoritmy účinnější, než kdyby byly použity samostatně.

Výsledků této bakalářské práce by se dalo využit například v rámci projektu RadegastXdb [9], který řeší nativní XML databázi. Nástroj pro kompresi vnitřního úložiště zde však chybí. Pro lepší kompresní poměry by bylo potřeba tuto práci do budoucna ještě rozšířit o automatické spouštění sémantických kompresorů.

A Návod k použití

Aplikace se dá spustit pomocí příkazového řádku, či ji lze spustit pomocí soubor nazvaného „xcom.exe“ a řídit se pokyny grafického rozhraní. Interakce s uživatelem probíhá pomocí příkazového řádku systému Windows. Uživatel komunikuje s aplikací pomocí zadání číselné hodnoty pro výběr vlastnosti, či zadá manuálně cestu k souboru, se kterým chce pracovat. Uživatel vždy ví, jaké jsou jeho momentální možnosti, jelikož aplikace uživatele v každém kroku jasně informuje o tom, co se po něm v daném okamžiku chce. Aby se předešlo nesmyslným vstupům uživatele, jsou vstupy ošetřeny tak, že pokud uživatel zadá něco chybně, bude o tom informován větou „Neplatna volba!“. Aplikace hovoří česky, avšak nevyužívá českou interpunkci díky možným problémům se znakovou sadou a špatném zobrazování některých písmen. Příklad takové interakce s aplikací je znázorněn na obrázku č. 13.

Pro spuštění algoritmů z příkazové řádky je potřeba zadat kromě cesty k souboru ještě další 4 parametry. Aplikace se tedy spouští sekvencí „xcom.exe [k/d] [algoritmus] [vstupní soubor] [výstupní soubor]“.
Do parametru [k/d] se zapíše:

- 1 pro kompresi.
- 2 pro dekompresi.

V parametru [algoritmus] je vyžadováno číslo algoritmu z následujícího seznamu:

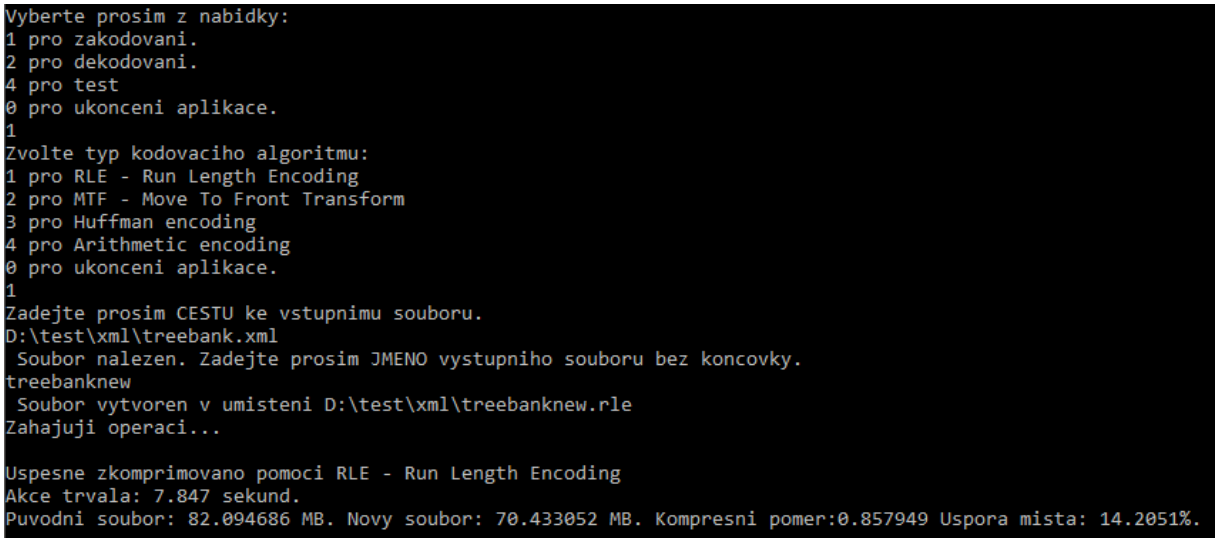
- 1 – Run lenght encoding
- 2 – Move To Front transform
- 3 – Huffmanovo kódování
- 4 – Lempel-Ziv 1977
- 5 – XMill
- 6 – XGrind

Jako [vstupní soubor] musí být uvedena celá cesta ke vstupnímu souboru.
Jako [výstupní soubor] stačí napsat pouze název souboru. Ten se vytvoří s příponou příslušící jednotlivým kompresním algoritmům ve stejném adresáři, ve kterém se nalézá vstupní soubor.

Pro spuštění dekomprese souboru „ahoj.xmi“ v umístění „D:\test“ pomocí algoritmu XMill do souboru „dekomprimovany.xml“ v umístění „D:\test“ bude příkaz následující:

xcom.exe 2 5 D:\test\ahoj.xmi dekomprimovany

Pokud program při spuštění metod XMill či XGrind hlásí chybu nenalezené knihovny, je třeba přiložený soubor „xerces-c_3_1.dll“ přesunout do složky „C:\Windows\System32“.



Obrázek 13: Uživatelské rozhraní.

Automatizovaný test

Ve složce test lze nalézt soubor „test.bat“. Tento soubor postupně spouští jednotlivé kompresní algoritmy a provádí kompresi a dekompresi XML dokumentu „kolekce.xml“. Zkomprimované a dekomprimované soubory jsou rozděleny do složek, které jsou pojmenovány podle názvu aplikovaného algoritmu. Výsledky komprese a dekomprese jsou uloženy do souboru „report.csv“.

Jelikož jsou data v tomto souboru rozdělena středníkem, pro lepší orientaci je doporučeno si tento soubor otevřít v programu MS Excel. Pro rozdělení dat klikněte na sloupec A. Dále klikněte na záložku Data > Text do sloupců > Další > zaškrtněte možnost „Středník“ > Dokončit.

Literatura

[1] *SAX [online]*, leden 2002. http://sax.sourceforge.net/.

[2] *The bzip2 home page [online]*, 1998 march. https://web.archive.org/web/19980704181204/http://www.muraroa.demon.co.uk/.

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml). *World Wide Web Journal*, 2(4):27–66, 1997.

[4] J. Cheng and W. Ng. Xqzip: Querying compressed xml using structural indexing. In *EDBT*, 2004.

[5] W. W. W. Consorcium. *Extensible Markup Language (XML) 1.0 (Fifth Edition) [online]*, leden 2000. https://www.w3.org/TR/xml/.

[6] W. W. W. Consorcium. *Extensible Markup Language (XML) 1.1 (Second Edition) [online]*, prosinec 2001. https://www.w3.org/TR/xml11/.

[7] M. Dipperstein. *LZSS (LZ77) Discussion and Implementation [online]*, listopad 2015. http://michael.dipperstein.com/lzss/.

[8] R. H. Google. *Loop Recognition in C /Java/Go/Scala*, září 2007. https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf.

[9] D. R. Group. *Projects - DBRG[online]*, březen 2015. http://db.cs.vsb.cz/Projects.

[10] J. Havel. *DOM – objektový model dokumentu [online]*, únor 2002. https://www.zive.cz/clanky/dom--objektovy-model-dokumentu/sc-3-a-104999/default.aspx.

[11] H. Liefke and D. Suciu. Xmill: an efficient compressor for xml data. In *ACM Sigmod Record*, volume 29, pages 153–164. ACM, 2000.

[12] J.-K. Min, M.-J. Park, and C.-W. Chung. Xpress: a queriable compression for xml data, leden 2003.

[13] I. Pavlov. *LZMA SDK (Software Development Kit) [online]*, červen 2015. http://www.7-zip.org/sdk.html.

[14] D. Salomon. *Data compression: the complete reference*. Springer Science & Business Media, 2004.

[15] B. Schaffner. *Send binary data in XML [online*, listopad 2002. https://www.techrepublic.com/article/send-binary-data-in-xml/.

[16] A. Schmidt. *XMark [online]*, květen 2001. https://web.archive.org/web/20180323014710/http://xml-benchmark.org/.

[17] T. B. Terriberry. *On the Overhead of Range Coders [online]*, červen 2009. `https://people.xiph.org/~tterribe/notes/range.html`.

[18] P. M. Tolani and J. R. Haritsa. Xgrind: a query-friendly xml compressor, 2002. pages 225–234.